Opening Polyhedral Compiler's Black Box

Lénaïc Bagnères / Oleksandr Zinenko
Inria and Univ. Paris-Saclay,
Orsay, France
lenaic.bagneres@inria.fr / oleksandr.zinenko@inria.fr

Stéphane Huot
Inria,
Lille, France
stephane.huot@inria.fr

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

Abstract

While compilers offer a fair trade-off between productivity and executable performance in single-threaded execution, their optimizations remain fragile when addressing compute-intensive code for parallel architectures with deep memory hierarchies. Moreover, these optimizations operate as black boxes, impenetrable for the user, leaving them with no alternative to time-consuming and error-prone manual optimization in cases where an imprecise cost model or a weak analysis resulted in a bad optimization decision. To address this issue, we propose a technique allowing to automatically translate an arbitrary polyhedral optimization, used internally by loop-level optimization frameworks of several modern compilers, into a sequence of comprehensible syntactic transformations as long as this optimization focuses on scheduling loop iterations. This approach opens the black box of the polyhedral frameworks, enabling users to examine, refine, replay and even design complex optimizations semi-automatically in partnership with the compiler.

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

Keywords Interactive Compilation; Loop Transformations

1. Introduction

Compiler/programmer interaction is a key tool to sustain high productivity and performance while developing applications for modern architectures. On one hand, compiler techniques may help programmers at rapidly writing correct codes through, e.g., automatic completion or as-you-write error detection. On the other hand, programmers may help compilers at generating efficient codes by providing additional information to complete their analyses through, e.g., options, specific keywords (such as `restrict`) or language extensions (such as OpenMP) supported by most compilers.

Most compiler optimizations are processed in an automatic way and/or are driven by heuristics offering little control to the developer. As a result, any lack of precision and analysis power may result in dramatic performance losses. Moreover, the compiler may diverge from user's intention and jeopardize manual optimizations. While existing interactive techniques focus on helping the programmer to write a code the compiler can handle [10, 12, 15], we present a novel approach to compiler feedback and control in the context of polyhedral frameworks, which power production compilers such as GCC [17], LLVM [11] or IBM XL [4].

Polyhedral frameworks operate on a mathematical representation of loop-based programs: each execution of a statement nested in loops is scheduled according to a system of (in)equalities. Once the code is *raised* to the polyhedral model, it may be restructured by modifying this system without any syntactic meaning until the final *code generation*. Thanks to its algebraic nature, the polyhedral model enables precise and deep analysis of loop-based programs that resulted in major advances in loop-level optimization over the last two decades [3, 7, 18]. However, polyhedral optimization remains a black box inside the compiler with no connection to the observable code transformations. In our work, we enable automatic translation from the state-of-the-art polyhedral representation to understandable syntactic transformation primitives and back.

Our paper brings two main contributions. First, we revisit the classical syntactic loop transformations (e.g., loop fusion or tiling) in the light of a new polyhedral formalism by expressing them as modifications of the scheduling relation only, ensuring their composability. Our transformation set is the first to allow representing arbitrary polyhedral optimizations. Second, we present an algorithm that translates scheduling relations to a sequence of syntactic loop transformations. These two contributions together virtually open the polyhedral framework's black box by making syntactic loop transformation the language for developer-compiler communication. The developer obtains precise feedback and full control over the polyhedral engine enabling him to create, analyze and refine loop-level optimizations.

2. Motivating Example

Polyhedral compilation frameworks provide state-of-the-art yet not flawless automatic optimization and parallelization of loop-based program parts. Our work aims at enabling an efficient interaction loop with such optimizers through sequences of comprehensible syntactic transformations. To illustrate our approach, we consider the main computational kernel of a conventional beamforming radar application shown in Fig. 1 and we parallelize it with the Pluto tool[1] [3]. The transformed code (Fig. 2) indeed contains parallel loops annotated with OpenMP directives. But surprisingly, on the target architecture[2], the original code takes 2.37 seconds while the transformed code takes 3.32 seconds, a slowdown.

```
                                                // L0
t = 0;                                            // S0
t_val = DBL_MIN;                                  // S1
for (i = 0; i < N; i++) {                         // L1
  a_i[i] = 0;                                       // S2
  a_r[i] = 0;                                       // S3
  for (j = 0; j < M; j++) {                       // L2
    a_r[i] += s_r[j]*m_r[i][j] - s_i[j]*m_i[i][j]; // S4
    a_i[i] += s_i[j]*m_r[i][j] + s_r[j]*m_i[i][j];}// S5
  val = a_r[i]*a_r[i] + a_i[i]*a_i[i];            // S6
  t = (val >= t_val)? (t_val = val, i) : t; }     // S7
```

Fig. 1: Conventional Beamforming Original Kernel

```
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  a_r[i] = 0;
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  for (j = 0; j <= M-1; j++)
    a_r[i] += s_r[j]*m_r[i][j] - s_i[j]*m_i[i][j];
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  a_i[i] = 0;
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  for (j = 0; j <= M - 1; j++)
    a_i[i] += s_i[j]*m_r[i][j] + s_r[j]*m_i[i][j];
t = 0;  t_val = DBL_MIN;
for (i = 0; i <= N - 1; i++) {
  val = a_r[i]*a_r[i] + a_i[i]*a_i[i];
  t = (val >= t_val)? (t_val = val, i) : t; }
```

Fig. 2: Pluto Automatically Generated Code

At this point, a user may either stick with the original sequential code or, if he is an expert, rework Pluto's output or use it as a feedback to manually optimize the original code.

We propose a new alternative to time-consuming and error-prone manual code modification: the polyhedral framework reports the sequence of loop transformations corresponding to the optimization for the user to review and to modify. The sequence corresponding to Pluto's optimization is shown in Fig. 4. Its detailed semantics is presented in Section 5. The user may notice that Pluto has been overly aggressive at loop distribution, reducing data locality and adding thread synchronization costs. Hence, he may simply modify the sequence to keep only the distribution that extracts the non-parallel part of the `L1` loop. Automatic semantics preservation check is then performed, and the final automatically generated code (Fig. 3) requires only 0.81 second to execute on the target architecture.

[1] Pluto 0.11.0 `http://pluto-compiler.sf.net`

[2] Experimental setup: 6-core Intel Xeon X5650 2.67GHz architecture, GNU GCC 4.8.2 compiler with options `-O3 -fopenmp`, and program parameters `N` and `M` set to 1000.

```
#pragma omp parallel for
for (i = 0; i <= N - 1; i++) {
  a_r[i] = 0;  a_i[i] = 0;
  for (j = 0; j <= M - 1; j++) {
    a_r[i] += s_r[j]*m_r[i][j] - s_i[j]*m_i[i][j];
    a_i[i] += s_i[j]*m_r[i][j] + s_r[j]*m_i[i][j]; } }
t = 0;  t_val = DBL_MIN;
for (i = 0; i <= N - 1; i++) {
  val = a_r[i]*a_r[i] + a_i[i]*a_i[i];
  t = (val >= t_val)? (t_val = val, i) : t; }
```

Fig. 3: Manually Refined Generated Code

Our approach is supported by (1) a complete mapping between classical and polyhedral transformations and (2) an algorithm to express a polyhedral optimization as a sequence of such transformations. It enables programmer-compiler partnership in code optimization that benefits from polyhedral representation and avoids manual code modification.

`distribute((2), 3)`	`reorder((), (4,5,2,0,1,3,6))`
`distribute((2), 2)`	`parallelize((0))`
`distribute((2), 1)`	`parallelize((1))`
`distribute((4, 0), 1)`	`parallelize((2))`
`distribute((4), 1)`	`parallelize((3))`

Fig. 4: Transformations Generated for Pluto Optimized Code

3. Polyhedral Compilation

Modern compilers perform multiple optimization passes to ultimately generate an efficient code. Loop nest optimization passes, which typically include loop vectorization and parallelization, often rely on the polyhedral model [4, 11, 17]. Its key aspect is to raise some program parts to an algebraic representation that enables both precise data dependence analysis and complex program restructuring. This model is applicable to parts of the program in which loop bounds, branch conditions and array subscripts are affine expressions of outer loop iteration variables and constant parameters. Despite these limitations, the model allows to capture major parts of compute-intensive loops used in scientific code eager for performance, for example the code in Fig. 1. Moreover, the model was extended to cover non-static control loops, up to full functions [2]. In this paper, we use the state-of-the-art *union of relations* abstraction for the polyhedral model [21]. It defines all components of the representation as unions of multidimensional relations that, contrary to the conventional scheduling functions, capture non-unit loop strides, disjunctive conditions and over-approximated data dependences.

Because loop boundaries and conditions that surround a statement are linear expressions, the *iteration domain* of that statement can be expressed as a system of linear inequalities. It defines a polyhedron in the multidimensional space. Integer points inside this iteration domain polyhedron represent a particular execution of the statement, or *statement instance*, in the surrounding loop nest. Their coordinates correspond to the loop iteration variables. For example the statement `S4` on the Fig. 1 is enclosed by two loops, first on `i` from `0` to `N`, second on `j` from `0` to `M`. Its iteration domain is thus two-dimensional on $\langle i, j\rangle$ and it is defined as shown in Fig. 5. Although the iteration domain is a set, it is treated as a degenerate relation without input dimensions for the sake of consistency with other components of the model.

$$\mathcal{D}_{S4}(\mathrm{N},\mathrm{M}) = \left\{ () \to \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{array}{l} 0 \le i < \mathrm{N} \\ 0 \le j < \mathrm{M} \end{array} \right\}$$

Fig. 5: Iteration Domain

A disjunction in the control structure condition results in an iteration domain described by a *union of relations* rather than by a single one. Each component of the union corresponds to a term of disjunction.

The order in which statement instances are executed is defined by the *scheduling relation*. It maps each point of the iteration domain (*input dimensions*) to a multidimensional logical date (*output dimensions*). In the target code, statement instances are executed according to the lexicographical order of the logical date dimensions. Output dimensions may carry semantic information useful for further steps, for example being parallel, vectorizable or unrolled. We also use a specific structure of output dimensions (Fig. 6): even output dimensions, denoted α, represent the resulting loop iteration variables, and odd output dimensions, denoted β, encode statement ordering and loop nesting (see Section 4.2).

If different instances of the same statement are to be executed in different order, the scheduling is described by a *union of relations*. Each relation in this union may have supplementary inequalities that limit its applicability to a particular subset of instances. In the following section, we discuss the constraints on the scheduling relations that allow them to remain applicable to any compatible domain.

$$\theta_{S4}(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{ll} \beta_0 & = 2 \\ \alpha_1 & = i \\ \beta_1 & = 2 \\ \alpha_2 & = j \\ \beta_2 & = 0 \end{array} \right\}$$

Fig. 6: Scheduling Relation

In the polyhedral model, a change in the scheduling relation corresponds to a program transformation. For example, to reverse the inner loop enclosing the statement `S4`, we can simply change the sign of i in the scheduling relation in Fig. 6. In this case, the iterations with higher values of i are scheduled before those with lower values, a loop reversal. By using unions of relations, we are able to encode in the scheduling the loop transformations that previously required iteration domain modifications, such as strip-mining and index-set splitting. Polyhedral optimizer's work in this case is to construct scheduling relations that reach the optimization objectives, e.g., parallelization or improved memory access locality, while ensuring that the transformation preserves the original program semantics. It functions by solving systems of inequalities that model every aspect of the program and of the optimization objective functions since there is no syntactic equivalent of the statement instance [3, 7, 18]. *It rarely, if ever, operates using classical loop transformations.*

Eventually, a code generation step builds a code that implements the new scheduling [1]. Only at this step the syntactic nature of the transformation may appear. However the model neither explains how the code was obtained, nor does give any control in case the heuristic failed or an expert user wants to suggest a refinement.

4. Analysis of Scheduling Relation

In order to establish a correspondence between scheduling relations and syntactic transformations, we discuss the limitations allowing to keep these schedulings applicable to any iteration domain within the polyhedral framework.

4.1 Scheduling Equivalence

Since the schedule represents the *order* of execution, the logical execution dates may take arbitrary numerical values as long as the order is preserved. We consider *equivalent* scheduling relations that yield the schedules with identical execution order, and we consider *equal* relations that yield the schedules with identical dates. Equal scheduling relations may include different (in)equalities, for example $((i,j) \to (\alpha_1,\alpha_2) : \alpha_1 + \alpha_2 = i + j \wedge \alpha_2 = j)$ is equal to $((i,j) \to (\alpha_1,\alpha_2) : \alpha_1 = i \wedge \alpha_2 = j)$ even though the equations are different. One may suggest a "normalized form" based on, e.g., Gaussian elimination, in which equal relations are defined by identical (in)equalities.

A change to the scheduling relation union that results in an equivalent schedule is called *equivalent transformation*, while the change resulting in an equal schedule is *invariant transformation*. Any algorithm operating within the polyhedral representation is free to perform *invariant* transformations for its needs. *Equivalent* transformations are allowed as long as no subsequent step operates on the numerical value of the date, for example the code generation algorithm may omit empty loop iterations by introducing a loop stride as in [1], but is not obliged to do so.

If scheduling relations are *equal* for all statements, so is the program scheduling, but it is not guaranteed for *equivalent* relations due to the relative nature of the definition.

4.2 Capturing and Recovering Lexical Order

We use a specific form of scheduling relations to capture the lexical order of statements, first suggested by Feautrier [8] and common to many polyhedral frameworks [5, 9]. Each odd *output* dimension, also called β-dimension, in the scheduling relation is a constant representing the order of statements in the loop of corresponding depth.

A β-vector of a scheduling relation is a vector composed from constant values of the β-dimensions in this relation, $(2, 2, 0)$ for the Fig. 6. β-vectors are unique across all relations in the scheduling and must not be prefixes of other β-vectors. On the contrary, one statement has as much β-vectors as it has relations in its scheduling union.

Like all *output* dimensions, β-dimensions define the order of execution rather than exact statement positions. For the sake of simplicity, we assume all β-vectors to be in the *normal form* with minimal non-negative values on each dimension that preserve the order. For example, the normal form of the β-vector set $\{(0,1),(2,1)\}$ is $\{(0,0),(1,0)\}$. The *β-normalization* is an *equivalent* transformation that does not affect other dimensions.

When the polyhedral analysis is applied to the syntactical representation, such as the source code or a syntax tree, we use a polyhedral raising tool, such as *Clan* [3], that generates normalized β-vectors for each scheduling relation. Although polyhedral optimizers do not necessarily respect the β-dimensions structure, we may recover them thanks to their semantics by recreating the loop structure with the code generation algorithm, e. g., CLooG [1], and then assign the β-vectors respecting the lexical order [8].

4.3 Ensuring Scheduling Validity

All transformation-related information is embedded into the scheduling relation, leaving the iteration domain immutable and allowing us to (1) reapply the precomputed transformation to domains of different shape and size, (2) delay semantics preservation checks until after the transformation. To ensure these properties, the scheduling relation must guarantee the existence of a unique integer execution date for each point in any domain by respecting the following constraints.

Equal input dimensionality – all relations in the scheduling union define the logical execution dates for the same iteration domain. We thus require all relations to have the number of *input dimensions* d equal to the dimensionality of the iteration domain. We refer to such relations as *d-compatible*.

Schedule existence – any *d-compatible* scheduling relation union must define a date for any d-dimensional integer point. An individual relation may include inequalities that restrict its applicability to the part of the iteration domain as long as the whole union maps the entire integer space $\mathbb{Z}^d$ to the corresponding logical dates.

Logical date integrality – all dates defined by the scheduling relation must be integer. Considered separately, the equations in the scheduling relation definition form a linear system with two logically distinct groups of unknowns that correspond to *input* or *output* dimensions. Solving this system for either of the groups results in the *input* or *output* form respectively where one dimension group is expressed as linear functions of another group. Linear functions with integer coefficients in the definition of the output dimensions guarantee date integrality for any domain.

If the scheduling relation has more output dimensions than input dimensions, some of the output dimensions do not have an *explicit* definition in terms of input dimensions. We allow such *implicitly defined* dimensions as long as they are bounded by inequalities, in which case the logical date for a statement instance comprises all the integer points within these bounds.

Logical date uniqueness – each instance in the d-dimensional iteration domain must be executed exactly once and its logical execution date should remain unique throughout the entire schedule of the program. Thanks to unique β-vectors in each relation, it is sufficient to ensure date uniqueness within an individual relation.

We refer to the scheduling relation unions that respect all these conditions as *globally valid* in the sense they can be used to schedule any *compatible* domain. When a relation violates one of these constraints, it may still remain *conditionally valid* if it respects them for domains that satisfy particular conditions. Finally, a relation is *invalid* if there is no domain for which it respects all the conditions. To avoid analyzing the iteration domain, scheduling relations should remain *globally valid* throughout the optimization process.

For example a one-dimensional scheduling that assigns execution dates only for the first ten iterations is conditionally valid since it fails to schedule all iterations of a larger domain (existence constraint). A relation with implicitly defined dimension bounded by two different explicitly defined dimensions is invalid since it assigns multiple dates for the same point (uniqueness constraint). Even though some polyhedral frameworks allow fractional *explicitly defined* dimensions, we consider them invalid (integrality constraint). We suggest they do not improve expressive power of the scheduling as an *equivalent* transformation that multiplies all fractional dates by a constant factor makes them integer.

Parallelism may be expressed by adding "parallel" semantics to a dimension, meaning that the actual dates on this dimension may be ignored. Assigning identical dates as a means to express parallelism, in addition to being invalid, does not express all forms of parallelism.

[3] `http://icps.u-strasbg.fr/~bastoul/development/clan/index.html`

5. Syntactic Transformations to Relations

The polyhedral scheduling relation abstraction is too complex to be used directly while classical loop transformation directives like *tile* or *fuse* or *skew* offer decent understandability. Combining them, programmers would benefit both from the exact instance-wise data dependence analysis and the automatic code generation of polyhedral frameworks and from the concise expressivity of directives.

We present a new revisiting of classical loop transformations in the polyhedral model, called after its implementation *Clay*. Contrary to existing approaches that also expose high-level transformation directives on top of a polyhedral engine such as UTF [13], URUK [9] or CHiLL [5], it is based on the more general scheduling relation abstraction rather than scheduling functions. It also relaxes unimodularity and invertibility limitations present in the previous work as long as relations respect the conditions listed in Section 4.3. *Clay* embeds the complete scheduling information in a single relation union per statement, even for the transformations previously requiring iteration domain modifications (StripMine and IndexSetSplit), thus removing the previously necessary intermediate data dependence graph updates and checks.

Clay relies on the loop-capturing scheduling structure with β-dimensions described in Section 4.2. Even scheduling dimensions, denoted α, define the execution order within the loop. Hence, the general form of the scheduling output dimension vector is $\vec{\sigma} = (\beta_0, \alpha_1, \beta_1, \alpha_2 \ldots, \beta_{n-1}, \alpha_n, \beta_n)^T$. The original scheduling of a program can be constructed as follows. For a statement enclosed in an n-dimensional loop, we introduce $(2n+1)$ logical time dimensions. The β_i dimension denotes the lexical position of the statement at the i^{th} nesting level, while the α_i is equal to the i^{th} dimension of the iteration space.

To express classical loop transformations using the relation formalism and our scheduling relation structure, we use notations and operators shown in Figure 7. They are used to represent specific subsets of scheduling union components and relation dimensions. The notion of β-prefix is paramount. It is used to select specific subsets of relations to be affected by the transformation. From a syntactic point of view, a β-prefix addresses a specific loop (or, equivalently, the set of statements enclosed in that loop). The empty vector is a particular β-prefix used to select all scheduling relations, or, from a syntactic point of view, the root of the program. In order to apply a transformation to an individual statement inside the loop, this statement should be extracted into a loop with distinct β-prefix and fused back with its original after transformation. Although one may suggest transformation primitives that operate on individual statements, they contradict to the idea of using *loop* transformations to express polyhedral schedules.

θ	scheduling relation $\theta(\vec{p}) = \bigcup_i \mathcal{T}_i$
$\mathcal{T}$	scheduling union component
$\vec{p}$	vector of program parameters
$\vec{\iota}_{\mathcal{T}}$	vector of input dimensions of $\mathcal{T}$
$\vec{\sigma}_{\mathcal{T}}$	vector of output dimensions of $\mathcal{T}$
$\vec{\alpha}_{\mathcal{T}}$	α-vector of $\mathcal{T}$, i.e., the vector of even dimensions of $\vec{\sigma}_{\mathcal{T}}$
$\vec{\beta}_{\mathcal{T}}$	β-vector of $\mathcal{T}$, i.e., the vector of odd dimensions of $\vec{\sigma}_{\mathcal{T}}$
$\vec{\rho}$	β-prefix, if empty it corresponds to the root level
$\vec{\alpha}_{\mathcal{T},i}$	symbolic i^{th} element of any α-vector
$\mathcal{T}_*$	set of all union components of all scheduling relations
$\mathcal{T}_{\vec{\rho}}$	subset of $\mathcal{T}_*$ restricted to union components such that $\vec{\rho}$ is a β-prefix, i. e. $\vec{\beta}_{\mathcal{T},1\ldots\dim\vec{\rho}} = \vec{\rho}$
$\mathcal{T}_{\vec{\rho},next}$	subset of $\mathcal{T}_*$ such that $\vec{\rho}_{1\ldots\dim\vec{\rho}-1}$ is the β-prefix and $\beta_{\mathcal{T},\dim\vec{\rho}} = \vec{\rho}_{\dim\vec{\rho}} + 1$; denotes all scheduling relations inside the loop that is immediately following the one defined by $\vec{\rho}$
$\mathcal{T}_{\vec{\rho},>}$	subset of $\mathcal{T}_*$ restricted to union components such that $\vec{\rho}_{1..\dim\vec{\rho}-1}$ is the β-prefix and $\beta_{\mathcal{T},\dim\vec{\rho}} > \vec{\rho}_{\dim\vec{\rho}}$; denotes all scheduling relations for statements and loops following the one defined by $\vec{\rho}$ within the same enclosing loop
$a \mapsto b$	substitution operator: replaces all occurrences of a with b throughout scheduling relations

Fig. 7: Notations and Operators used in the Clay Formalism

5.1 Revisiting Classical Transformations in Clay

We revisit classical loop transformations [22] using the union of relations abstraction and notations in Fig 7. Each transformation is presented as a primitive, arguments of which can be integers, integer vectors or affine constraints satisfying the preconditions in order to enforce *global validity* of the scheduling after the transformation if the initial scheduling respected it (Fig. 8).

Reorder($\vec{\rho}$, $\vec{v}$) reorganizes statements and loops inside the loop defined by $\vec{\rho}$ according to the vector $\vec{v}$. The i^{th} element of $\vec{v}$ corresponds to the new position of i-th statement (loop), sorted lexically.

FuseNext($\vec{\rho}$) fuses the loop corresponding to $\vec{\rho}$ with its direct successor. Keeps the original order of nested loops and statements.

Distribute($\vec{\rho}$, n) distributes statements in the loop between two succeeding loops, the first containing first n statement of the original loop, i. e. those with $\vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1} < n$

Shift($\vec{\rho}$, i, $amount$) moves all instances of statements with β-prefix $\vec{\rho}$ in the iteration space by constant (parametric) $amount$ in the i^{th} output loop. For relations with only *explicitly defined* dimensions, it may be performed by modifying the parameters and the constant in the definition by $-amount$. Substitution is required to preserve inequalities, including those *implicitly defining* the dimension.

Transformation	Effect, Condition
REORDER $(\vec{\rho}, \vec{v})$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}},\ \ \vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1} \leftarrow \vec{v}_{\vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1}}$ **where** $\dim\vec{v} = \max_{\mathcal{T}\in\mathcal{T}_{\vec{\rho}}}\left(\vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1}\right) + 1\,; \forall i\ \ 1 \le i \le \dim\vec{v},\ \ 0 \le (\vec{v})_i \le \dim\vec{v} - 1\,; \forall i, j\ \ i \ne j,\ \ (\vec{v})_i \ne (\vec{v})_j$
FUSENEXT $(\vec{\rho})$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},next}, \vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1} + \max_{\mathcal{T}\in\mathcal{T}_{\vec{\rho}}}\left(\vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1}\right) + 1\,; \forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},>}, \vec{\beta}_{\mathcal{T},\dim\vec{\rho}} \leftarrow \vec{\beta}_{\mathcal{T},\dim\vec{\rho}} - 1$ **where** $\exists \vec{\beta}_{\mathcal{T}} : (\vec{\beta}_{\mathcal{T}})_{1..\dim\vec{\rho}-1} = (\vec{\rho})_{1..\dim\vec{\rho}-1}$ and $\vec{\beta}_{\mathcal{T},\dim\vec{\rho}} = (\vec{\rho})_{\dim\vec{\rho}} + 1$ and $\dim\vec{\beta}_{\mathcal{T}} > \dim\vec{\rho}$
DISTRIBUTE $(\vec{\rho}, n)$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}} : \vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1} < n,\ \ \vec{\beta}_{\mathcal{T},\dim\vec{\rho}} \leftarrow \vec{\beta}_{\mathcal{T},\dim\vec{\rho}} + 1$; and normalize β-vectors as in Section 4.2. **where** $1 \le n < \max_{\mathcal{T}\in\mathcal{T}_{\vec{\rho}}}\left(\vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1}\right)$
SHIFT $(\vec{\rho}, i, amount)$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T},i} \mapsto \vec{\alpha}_{\mathcal{T},i} + amount$ **where** $1 \le i \le \dim\vec{\rho} \le \dim\vec{\alpha}_{\mathcal{T}}$ and $amount = \vec{v}\cdot\vec{p} + C, \vec{v} \in \mathbb{Z}^{\dim\vec{p}}, C \in \mathbb{Z}$
SKEW $(\vec{\rho}, i, k)$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T},\dim\vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},\dim\vec{\rho}} + k\cdot\vec{\alpha}_{\mathcal{T},i}$ **where** $1 \le i \le \dim\vec{\alpha}_{\mathcal{T}} \wedge i \ne \dim\vec{\rho} \wedge k \in \mathbb{Z}^*$
REVERSE $(\vec{\rho})$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T},\dim\vec{\rho}} \mapsto -\vec{\alpha}_{\mathcal{T},\dim\vec{\rho}}$ **where** $\dim\vec{\rho} < \dim\vec{\beta}_{\mathcal{T}}$
INTERCHANGE $(\vec{\rho}, i)$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}},\ \ \vec{\alpha}_{\mathcal{T},i} \mapsto \vec{\alpha}_{\mathcal{T},\dim\vec{\rho}} \wedge \vec{\alpha}_{\mathcal{T},\dim\vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},i}$ if both substitutions are done simultaneously **where** $1 \le i < \dim\vec{\rho}$
RESHAPE $(\vec{\rho}, i, k)$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T},\dim\vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},\dim\vec{\rho}} + k\cdot\vec{\iota}_{\mathcal{T},i}$ **where** Either $\alpha_{\mathcal{T},i}$ or $\alpha_{\mathcal{T},\dim\vec{\rho}}$, or both are *implicitly defined*. Alternatively, for *explicitly defined* $\alpha_{\mathcal{T},i} = \vec{v}\cdot\vec{\iota}^{T} + f_1(\vec{p}) + C_1$ and $\alpha_{\mathcal{T},\dim\vec{\rho}} = \vec{w}\cdot\vec{\iota}^{T} + f_2(\vec{p}) + C_2, \left(\nexists d : d = \frac{\vec{v}_j}{\vec{w}_j}\forall j \ne i\right) \vee \left(\frac{\vec{v}_j}{\vec{w}_j + k\vec{v}_j} \ne d\right)$. For *explicitly defined* $\alpha_{\mathcal{T},\dim\vec{\rho}} = w\cdot\vec{\iota}_i + f_2(\vec{p}) + C_2, w \ne -k,$
INDEXSETSPLIT $(\vec{\rho}, constraint)$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \mathcal{T} \mapsto \mathcal{T}'\cup\mathcal{T}''; \mathcal{T}' = \mathcal{T}\cap constraint\,, \mathcal{T}'' = \mathcal{T}\cap\neg constraint\cap\vec{\beta}_{\mathcal{T}'',\dim\vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T}'',\dim\vec{\rho}+1} + \max_{\mathcal{T}\in\mathcal{T}_{\vec{\rho}}}(\vec{\beta}_{\mathcal{T},\dim\vec{\rho}+1}) + 1$ **where** $constraint = \vec{u}\times\vec{\alpha}_{\mathcal{T}} + \vec{v}\times\vec{\iota}_{\mathcal{T}} + \vec{w}\times\vec{p} + C\,; \vec{u} \in \mathbb{Z}^{\dim\vec{\alpha}}, \vec{v} \in \mathbb{Z}^{\dim\vec{\iota}}, \vec{w} \in \mathbb{Z}^{\dim\vec{p}}, C \in \mathbb{Z}$
COLLAPSE $(\vec{\rho})$	$\forall \mathcal{T}', \mathcal{T}'' \in \mathcal{T}_{\vec{\rho}} : \mathcal{T}' = \mathcal{T} \cap constraint \wedge \mathcal{T}'' = \mathcal{T} \cap \neg constraint \wedge \quad \vec{\beta}_{\mathcal{T}',1\ldots\dim\vec{\rho}} = \vec{\beta}_{\mathcal{T}'',1\ldots\dim\vec{\rho}} = \vec{\rho} \wedge \vec{\beta}_{\mathcal{T}',\dim\vec{\rho}+1} + 1 = \vec{\beta}_{\mathcal{T}'',\dim\vec{\rho}+1}\ (\mathcal{T}' \bigcup \mathcal{T}'') \mapsto \mathcal{T}$ **where** $constraint = \vec{u}\cdot\vec{\alpha}_{\mathcal{T}} + \vec{v}\cdot\vec{\iota}_{\mathcal{T}} + \vec{w}\cdot\vec{p} + C\,, \vec{u} \in \mathbb{Z}^{\dim\vec{\alpha}}, \vec{v} \in \mathbb{Z}^{\dim\vec{\iota}}, \vec{w} \in \mathbb{Z}^{\dim\vec{p}}, C \in \mathbb{Z}$
GRAIN $(\vec{\rho}, i, g)$	$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \forall$(in)equality $\left(\vec{u}\cdot\vec{\alpha}^{T} + f(\vec{\iota}, \vec{p}) + C \ge 0\right) \in \mathcal{T} : \vec{u}_i \ne 0,$ replace it by $\left(g\vec{u}\cdot\vec{\alpha}^{T} - (g-1)\vec{u}_i\vec{\alpha}_i + g\cdot f(\vec{\iota}, \vec{p}) + gC \ge 0\right)$ **where** $1 \le i \le \dim\vec{\rho} \le \dim\vec{\alpha}_{\mathcal{T}}$ and $g \ge 1$
DENSIFY $(\vec{\rho}, i)$	$\exists g \in \mathbb{N} : g > 1, \forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \forall$(in)equality $\left(g\vec{u}\cdot\vec{\alpha}^{T} - (g-1)\vec{u}_i\vec{\alpha}_i + g\cdot f(\vec{\iota}, \vec{p}) + gC \ge 0\right) \in \mathcal{T}, \vec{u} \in \mathbb{Z}^{\dim\vec{\alpha}}, C \in \mathbb{Z}, f$ is an integer linear function, replace this (in)equality by $\left(\vec{u}\cdot\vec{\alpha}^{T} + f(\vec{\iota}, \vec{p}) + C \ge 0\right)$ **where** $1 \le i \le \dim\vec{\rho} \le \dim\vec{\alpha}_{\mathcal{T}}$
STRIPMINE $(\vec{\rho}, size)$	$i \leftarrow \dim\vec{\rho}\,; \forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \mathcal{T} \leftarrow ((\uparrow_i (\mathcal{T})) \cap (size\cdot\vec{\alpha}_{\mathcal{T},i} \le \vec{\alpha}_{\mathcal{T},i+1} \le size\cdot\vec{\alpha}_{\mathcal{T},i} + size - 1))$ with unary operator $\uparrow_i (\mathcal{T})$ inserts a new α-dimension and a new β-dimension before the ith α-dimension **where** $size \in \mathbb{N}$
LINEARIZE $(\vec{\rho}, i)$	$i \leftarrow \dim\vec{\rho}\,; \forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \mathcal{T} \leftarrow (\downarrow_i (\mathcal{T}))$ with unary operator $\downarrow_i (\mathcal{T})$ removes the α-dimension and the β-dimension before the $i+1$th α-dimension and removes all (in)equations which contains the old ith α-dimension **where** $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, (size\cdot\vec{\alpha}_{\mathcal{T},i} \le \vec{\alpha}_{\mathcal{T},i+1} \le size\cdot\vec{\alpha}_{\mathcal{T},i} + size - 1) \subset \mathcal{T}$
PARALLELIZE $(\vec{\rho}, i)$	*Add semantic information* **where** $1 \le i \le \dim\vec{\rho} \le \dim\vec{\alpha}_{\mathcal{T}}$

Fig. 8: *Clay* Transformation Directives allow for encoding an arbitrary schedule

SKEW($\vec{\rho}$, i, k) makes the loop iterator at depth $\dim\vec{\rho}$ traverse the values of the loop iterator at depth i with a coefficient k (*skew factor*). This operation takes into account the *output* dimension, i. e. the ith loop iteration variable in the transformed code.

REVERSE($\vec{\rho}$) reverses the iteration order of the loop at depth $\dim\vec{\rho}$ for all statement instances with β-prefix $\vec{\rho}$.

INTERCHANGE($\vec{\rho}$, i) in a loop nest including statements with β-prefix $\vec{\rho}$, swaps the loop at depth i with the loop at depth $\dim\vec{\rho}$.

RESHAPE($\vec{\rho}$, i, k) reshapes the iteration space so that the loop iterator at depth $\dim\vec{\rho}$ depends on the *original* loop iterator at depth i with a coefficient k (*reshape factor*). This reshape operation takes into account the *input* dimension, i. e. the ith loop iteration variable as it was present in the original code. Useful for expressing complex transformations, such as skewing inner loop by a fraction of the outer loop, preserving the number of points. The conditions prevent originally *explicitly defined* dimensions from becoming linearly dependent or constant, which would violate the *scheduling existence* constraint since the transformed loop would no longer fully traverse explicitly associated input dimensions.

INDEXSETSPLIT($\vec{\rho}$, $constraint$) replaces every scheduling relation with β-prefix $\vec{\rho}$ by a union of two disjoint relations depending on the $constraint$ and having unique β-vectors; further transformations may target either relation.

COLLAPSE($\vec{\rho}$) squashes immediately following statements with β-prefix $\vec{\rho}$ scheduled identically by a disjoint pair of relations, replacing it with a single relation.

GRAIN($\vec{\rho}$, i, $grain$) changes, for each statement with β-prefix $\vec{\rho}$, the number of iterations n between two consecutive executions of the statement along the $\vec{\alpha}_{\mathcal{T},i}$ to $n \times grain$.

DENSIFY($\vec{\rho}$, i) removes, for each statement with β-prefix $\vec{\rho}$, the gap between two consecutive executions of the statement along the $\vec{\alpha}_{\mathcal{T},i}$.

STRIPMINE($\vec{\rho}$, $size$) decomposes the loop at depth $\dim\vec{\rho}$, for all statement instances with β-prefix $\vec{\rho}$, into two nested loops such that, for each iteration of the first loop, the second loop iterates over a chunk of at most $size$ consecutive iterations of the original loop.

LINEARIZE($\vec{\rho}$, i) integrates iterations of the nested loop into the host loop by extending its iteration variable span. This transformation remains *globally valid* only if applied to *implicitly defined* dimensions with non-parametric bounds, e. g. a dimension created by *stripmine*, since it effectively multiplies loop bounds.

PARALLELIZE($\vec{\rho}$, i) is an example of pseudo-transformation adding semantic information to the i^{th} output dimension of scheduling relations for each statement with β-prefix $\vec{\rho}$. Other semantic information may allow code generator to unroll loops or introduce vector operations.

For our motivating example, the user may either reorder parallel loops and fuse them together applying the transformations listed in Fig. 9,left to the transformed code in Fig. 2, or manually optimize the original code in Fig. 1 with transformations listed in Fig. 9,right.

5.2 Transforming between Arbitrary Scheduling Relations in Clay

Clay transformation set was designed to enable converting any *globally valid* scheduling to any other *globally valid* scheduling. This conversion implies changing both α, involved in equations and inequalities, and β-dimensions arbitrarily within the limits of *global validity* described above.

```
reorder((), (0,2,1,3,4,5,6))
fuse_next((0))
fuse_next((0))
fuse_next((0))
fuse_next((0, 2))
```

```
distribute((2), 3)
reorder((), (1,2,0,3))
reorder((0), (1,0,2))
parallelize((0))
```

Fig. 9: (left) transformations added by the user instead of rewriting the code; (right) optimizing transformation sequence created from scratch

Changing equations – we may transform the scheduling relation to its *output* form as described in Section 4.3. After this equivalent transformation, each *explicitly defined* dimension will be involved in exactly one equation (its definition) of the form $\alpha_{explicit} = \vec{u}\cdot\vec{\alpha}_{implicit}^{T} + \vec{v}\cdot\vec{\iota}^{T} + \vec{w}\cdot\vec{p}^{T} + C$. RESHAPE transformation allows to arbitrarily change coefficients in $\vec{v}$ and SHIFT coefficients in $\vec{w}$ and C. Non-zero $\vec{u}$ coefficients may appear after SKEW by an implicitly defined dimension replacing the unique appearance of $\alpha_{explicit}$ by $(\alpha_{explicit} - u\cdot\vec{\alpha}_{implicit})$. It suffices to take a difference between the existing and the target value before applying the transformation. Note that SKEW preconditions forbid substituting an *implicitly defined* dimension by a linear form involving an *explicitly defined* dimension that results in breaking the definition of the latter. Since the output form contains all the equations, we are able to change the equations arbitrarily with *Clay*.

Changing inequalities – *existence* and *uniqueness* constraints significantly reduce the structure of inequalities defining the scheduling relation. An arbitrary inequality makes the scheduling *conditionally valid* unless other relations in the scheduling union cover the remainder of the iteration domain. INDEXSETSPLIT adds an arbitrary inequality and enforces the *existence* constraint. Sequencing such transformations allows to build an arbitrary disjoint scheduling union. COLLAPSE transformation, on the other hand, joins two disjoint parts preserving domain coverage. A combination of these transformations may be used to redistribute domain points between scheduling relations. Implicitly defined dimensions are another source of inequalities. However, the only dimensions that enforce *uniqueness* for all compatible domains, are those created to express integer division as suggested by Bastoul [1]. STRIPMINE allows to create such dimensions that do not omit or duplicate instances, and LINEARIZE allows to undo it. Inequalities that are not involved in extra relations or dimensions render the scheduling *conditionally valid.* Hence *Clay* transformations suffice to express globally valid operations on inequalities.

Changing β-vectors – statement position encoded by β-vectors may be presented as an ordered forest. Each tree in the forest defines a loop nest. Nodes in the tree correspond to loops, and leaves correspond to statements. The depth of the leaf corresponds to the number of loops surrounding the respective statement. INDEXSETSPLIT and COLLAPSE transformation allow to increase or decrease the set of leaves. DISTRIBUTE transformation allows to split a node at any level into two separate nodes, each of which retains part of the original node's leaves. Applying this transformation to all nodes that have more than one child repeatedly will result in the forest of degenerate trees, where each node has exactly one child. It allows to reduce the lexicographic ordering of β-vectors into the simple ordering of first components of the β-vectors and use REORDER once to establish an arbitrary total order. At this point, one may add or remove nodes using *StripMine* or *Linearize*. Finally, FUSENEXT transformation allows to either merge the tree roots, or merge adjacent nodes inside one tree so that children of both nodes become children of the new merged node. Being applied to the forest of degenerate trees, it allows to establish any parent-child links. Therefore *Clay* allows to modify arbitrarily the number of leaves, nodes and the parent-child links that, together, fully define the forest. When β-forest represents a real program, all transformations are only applicable if the resulting scheduling remains *valid*.

5.3 Discussion of the Transformation Set

Redundancy – analyzing the transformation effects, one may notice that *Reverse* and *Grain* are, in fact, variations of *Skew* of a dimension by itself with coefficients -2 and $(k-1)/k$. Except these cases, *Skew* by itself results in fractional explicit definitions violating the *schedule existence* requirement. Since *Reverse* and *Grain* feature substantially different semantics and own preconditions, they were made transformations on their own.

Skew compared to Reshape – while *Skew* by an explicitly defined dimension may be expressed as a sequence of *Reshapes* involving coefficients in this explicit definition, *Skew* by an implicitly defined dimension remains a specific transformation on its own. It also offers powerful semantics for the end user.

Dimension independence – the corollary of *schedule existence* and *integrality* constraints requires explicitly defined dimensions to be linearly independent. This results in *Reshape* and *Skew* transformations having no effects beyond involved dimensions and makes them easily composable.

Higher-level transformations – higher-level syntactic transformations may be obtained combining *Clay* transformations. For instance, *tiling* is a classical composition of *Strip-Mine* and *interchange* while *rotation* transformation that aligns a dependence vector along certain loop corresponds to the composition of *grain* and *reshape*.

Beyond global validity – in order to avoid analyzing iteration domain, *Clay* requires all transformations to be *globally valid*. However, this constraint may be relaxed in the future by explicitly computing the validity conditions and verifying the domain against them before transformations.

6. Relations to Syntactic Transformations

Having established the mapping between loop transformations and unions of scheduling relations, we may propose an algorithm that represents such a union as a sequence of loop transformations. We exploit the properties of *Clay* transformations and our specific relation structure as described in Sections 6.1, 6.2 in order to create the final algorithm.

6.1 Detecting Complementary Transformations

Clay features pairs of complementary transformations that ensure invertibility as shown in Table 1. In each pair, one transformation has less arguments as it deduces them from its preconditions to properly undo the effect. We leverage this property in procedure COMPLEMENTARY that searches for transformations with smaller arity in both the original and transformed schedulings. If preconditions for the transformation with less arguments are met by the transformed relation, it means that the complementary transformation was applied to the original scheduling with arguments deducible from preconditions. For example, COLLAPSE identifies the separation inequality required for INDEXSETSPLIT. In order to support COLLAPSE that operates on a pair of scheduling relations, the procedure iterates over all pairs of relations instead of individual relations.

Table 1: Inverse transformations take only dimension index i as non-deducible parameter, if any.

Direct	Inverse	Deducible	Others
INDEXSETSPLIT	COLLAPSE	*condition*	$\vec{\rho}$
STRIPMINE	LINEARIZE	*size*	$\vec{\rho}$, i
GRAIN	DENSIFY	*grain*	$\vec{\rho}$, i
DISTRIBUTE	FUSENEXT	i	$\vec{\rho}$

As Table 1 suggests, several inverse transformation require dimension index as an non-deducible argument. We may iterate through all possible values, verifying precondition each time, since they are limited by the nesting of the loop in the source code that rarely exceeds 10.

6.2 Aligning Relations and Matching Beta-vectors

Given an initial and a target β-vector identifying the relation, we may compute a transformation sequence converting the former to the latter by first separating the statement in question into a loop that does not contain any other statements at a given depth, then reordering loops and fusing back if necessary following the algorithm 1. This algorithm operates on the entire set of statements, recursing by β-prefix depth. After each recursive step is completed, the values of β-vectors with the given prefix are equal to the target ones until the current depth. Note that after fusion, all the β-vectors that do not belong to the target prefix are immediately split away.

Algorithm 1 requires knowing the one-to-one mapping between original and target β-vectors for each relation. Yet scheduling unions may not have the same structure: INDEXSETSPLIT may have changed the union cardinality and STRIPMINE may have changed each relation dimensionality. We propose to add dummy transformations that will match the scheduling structure, INDEXSETSPLIT with always false condition and pseudo-STRIPMINE that creates a constant dimension. We then match relations following the order in which they appear in the union. Even if the optimizer breaks this order, we will find a (longer) transformation sequence by changing definitions arbitrarily. Once an actual INDEXSETSPLIT or STRIPMINE transformation is found, it replaces the dummy one in the sequence.

Algorithm 1: MATCHBETAS

```
1  let $\vec{\rho}$ be the current $\beta$-prefix ;
2  let $d = \dim \vec{\rho} + 1$ ;
3  let $n = \max_{\mathcal{T} \in \theta} \vec{\beta}_d$ where $\vec{\beta}_d$ is the $\beta$-vector of $\mathcal{T}$ ;
4  let $n' = \max_{\mathcal{T}' \in \theta'} \vec{\beta}'_d$ where $\vec{\beta}'_d$ is the $\beta$-vector of $\mathcal{T}'$ ;
5  while $\exists \vec{\beta}, \vec{\beta}' : \vec{\beta}_d \neq \vec{\beta}'_d$, $\vec{\beta}_d \to \min$ do
       /* split away until depth d */
6      for $i = \dim \vec{\beta}$ downto $d$ do
7          REORDER($\vec{\beta}_{1\dots(i-1)}$, put $\vec{\beta}_d$ last) ;
8          DISTRIBUTE($\vec{\beta}$, $i$) ;
9      if $\vec{\beta}_d \leq n$ then
10         REORDER($\vec{\beta}_{1\dots(d-1)}$, put $\vec{\beta}_d$ right after $\vec{\beta}'_d$) ;
11         FUSENEXT($\vec{\beta}_{1\dots d}$) ;
           /* split away betas that do not belong to
              the same transformed prefix */
12         foreach $\vec{\beta}, \vec{\beta}' : \vec{\beta}_d \neq \vec{\beta}'_d$ do
13             REORDER($\vec{\beta}_{1\dots(i-1)}$, put $\vec{\beta}_d$ last) ;
14             DISTRIBUTE($\vec{\beta}$, $i$) ;
15     else
16         REORDER($\vec{\beta}_{1\dots(d-1)}$, put $\vec{\beta}_d$ at $\vec{\beta}'_d$) ;
   /* n = n' at this point; recurse to sub-prefixes */
17 for $i = 0$ to $n$ do
18     recurse with $\vec{\rho} \leftarrow (\vec{\rho}_1, \vec{\rho}_2, \dots, \vec{\rho}_{\dim \vec{\rho}}, i)$ ;
```

6.3 Generating Transformation Sequence

Once the one-to-one matching between relations is established, we use the algorithm 2 to find the requested *Clay* transformation sequence. It builds on the observation that adding explicit dimension definitions corresponds to SKEW transformation and uses Gaussian elimination to coerce linear system partly defining the scheduling relation into a triangular form, after which it modifies individual coefficients with RESHAPE. After each transformation, it verifies if substitutions in inequalities enabled other transformations, thus performing structure-modifying transformation as soon as possible. It also leverages the fact that substituting implicitly-defined dimension will introduce equal coefficients into inequalities bounding it on both sides. When triangulating the transformed relation, the algorithm records inverse transformations (including SKEW with inverted coefficient) as though they have been applied to the original scheduling.

For our motivating example, the sequence of transformations between the original code (Fig. 1) and the optimized one (Fig. 2) generated by our algorithm consists only of loop distributions and reordering as shown in Fig. 4. The user may complete this sequence by their transformations (Fig. 9,left) in order to obtain the faster code.

Algorithm 2: WHITEBOXING

```
1  Transform all relations in both the original and
   optimized scheduling unions to the *output* form.;
2  repeat
3      restart iteration after each step with $\circlearrowright$;
4      MATCHBETAS ($\vec{\rho} \leftarrow ()$);
5      foreach statement $S$ do
6          foreach implicitly defined dimension $d$ do
7              COMPLEMENTARY(STRIPMINE) $\circlearrowright$;
8          COMPLEMENTARY(INDEXSETSPLIT) $\circlearrowright$;
9          foreach dimension $d$ do
10             COMPLEMENTARY(GRAIN) $\circlearrowright$;
11         foreach $\vec{\alpha}_i$, $i \rightarrow$ max, explicitly defined by
           $\{\alpha_i = \vec{v} \cdot \vec{\iota}^T + \vec{w} \cdot \vec{p}^T + C\}$ in $\mathcal{T}$ or in $\mathcal{T}'$ do
12             foreach $j \leftarrow 1..(i-1) : v_j \neq 0$ do
13                 let $x_j$ be the $v_j$ value in the explicit
                   definition of $\alpha_j$ ;
14                 use inverse transformation if $\in \theta'_S$ ;
15                 GRAIN($\beta_{\mathcal{T},1..i}$, lcm$(x_j, v_j)/v_j$) ;
16                 SKEW($\beta_{\mathcal{T},1..i}$, $j$, $-$lcm$(x_j, v_j)/x_j$) ;
17                 DENSIFY($\beta_{\mathcal{T},1..i}$) $\circlearrowright$;
18             if $\vec{v}_i < 0$ then
19                 REVERSE($\beta_{\mathcal{T},1..i}$) $\circlearrowright$;
20         foreach $\vec{\alpha}_i$, $\vec{\alpha}'_i$ both explicitly defined by
           $\{\alpha_i = \vec{v} \cdot \vec{\iota}^T + \vec{w} \cdot \vec{p}^T + C\}$ in $\mathcal{T}$, $\mathcal{T}'$ do
21             foreach $\vec{v}_j \neq \vec{v}'_j$ do
22                 RESHAPE($\beta_{\mathcal{T},1..i}$, $\vec{v}'_j - \vec{v}_j$) $\circlearrowright$;
23             repeat step 22 for $\vec{w}$ and $C$ with SHIFT ;
24         foreach $\vec{\alpha}_i$, $\vec{\alpha}'_i$ both implicitly defined by
           $\{\vec{u} \cdot \vec{\alpha}^T + \vec{v} \cdot \vec{\iota}^T + \vec{w} \cdot \vec{p}^T + C \geq 0\}$ in $\mathcal{T}, \mathcal{T}'$ do
25             foreach $\vec{u}_j \neq \vec{u}'_j$ do
26                 SKEW($\beta_{\mathcal{T},1..i}$, j, $\vec{u}'_j - \vec{u}_j$) $\circlearrowright$;
27             repeat step 26 for $\vec{v}$ with RESHAPE ;
28             repeat step 26 for $\vec{w}$ and $C$ with SHIFT ;
29 until no transformation happened in the loop;
```

6.4 Discussion of the Whiteboxing Algorithm

The proposed algorithm finds one sequence corresponding to the scheduling transformation out of infinite possibilities with no guarantees on its length or composition. Since a SKEW-like transformation may hinder COLLAPSE and LINEARIZE preconditions, the latter transformations are detected as soon as possible. It may result in more complex conditions that could have been avoided had the SKEW come first. Using triangulization allows to prefer SKEW, having stronger semantics, to RESHAPE at a cost of longer transformation sequences. In practice, Pluto rarely generates schedules with lots of non-zero coefficients thus reducing the sequence length. Future work should concentrate on introducing a post-processing step to simplify the sequence by reordering transformations and exploiting complementary transformations.

If the algorithm fails to detect an INDEXSETSPLIT condition, one may introduce a step that rendres two condition inequalities identical by modifying their parameters with SKEW and RESHAPE. In practice, we did not observe such situtation that corresponds to the algorithm leaving the main loop without having made the two schedulings equivalent.

We implemented the algorithm in Chlore project[4] reusing the *Clay* transformation set. We applied the transformation sequence recovery algorithm to Pluto-optimized codes from the Polybench suite[5]. Our implementation successfully recovered transformation sequences for all 30 benchmarks. It took on average 64 ms ($SD = 85$) per benchmark, ranging from 14 ms for `floyd-warshall` to 463 ms for `deriche` given that no optimization was applied to the implementation. The resulting transformation sequence contains 22 directives on average ($SD = 25$), ranging from 1 to 122 for the same benchmarks. Normalizing the number of transformations to the number of statements in the benchmark, the resulting sequence features a mean of 5 transformations per statements ($SD = 5.34$). These results suggest that, despite the computational complexity of the proposed transformation recovery algorithm, it may be used in an dialog interaction with the developer without substantial delays. However, more scrutiny is required to evaluate the developers' understanding of long transformation sequences or transformations of individual statements in large code blocks.

[4] https://periscop.github.io/chlore/

[5] Polybench/C 4.1, http://sourceforge.net/projects/polybench/

7. Interacting with a Polyhedral Compiler

Our approach introduces more interaction points between the optimizing polyhedral compiler and the developer. A typical standalone polyhedral compiler, such as Pluto, usually works on the source-to-source level presenting user with the automatically generated code while calling raising and code generation algorithms internally (see black and dashed red parts of the Fig. 10). From the user's point of view, the entire process happens in a single interaction step.

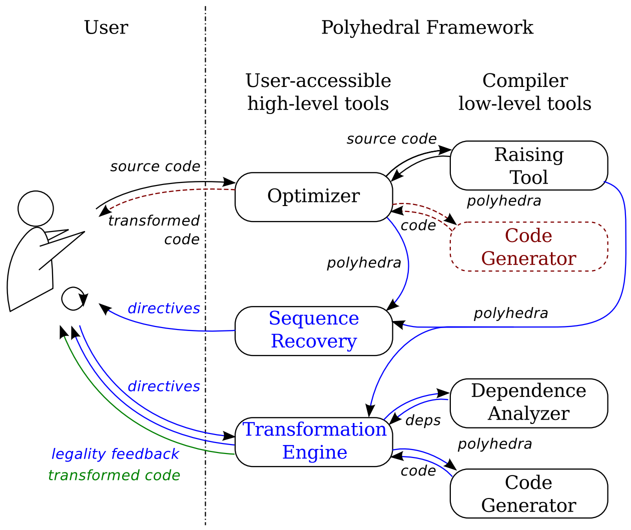

Fig. 10: Points of Interaction with the Polyhedral Framework: dashed parts were removed and lighter solid parts are added with *Clay*

By adding the transformation directive recovery algorithm and the transformation engine to the framework (light blue parts on the Fig. 10), we enable an iterative interaction: (1) the framework suggests an automatically generated transformation sequence for the user to review; (2) the user either accepts the sequence as is or modifies it before submitting to the transformation engine; (3) the latter reports on the semantics preservation and generates the final code if required. The user is free to keep modifying the sequence, repeating the two last steps and forming the interaction loop with the compiler where transformation directives and dependency graphs are used as a means for communication. They can also invoke the transformation engine directly, without relying on the automatic optimizer or completely ignoring the suggested transformation sequence, and continue interacting with the framework. We thus open a way for interaction between the developer and the polyhedral compiler, which can be leveraged in more efficient tools for semi-automatic compiler-assisted program optimization featuring visual representations [24] or a domain-specific language for optimizer fine-tuning.

8. Related Work

High-Level Loop Transformation Frameworks Several frameworks expose a high-level interface on top of a polyhedral engine, UTF [13] being arguably the very first of them. The URUK framework [9] features loop transformation sequences with unimodular schedules while CHiLL [5, 19] involves schedules defined by invertible relations thanks to dimensionality extension during preprocessing step. *Clay* relaxes both unimodularity and invertibility constraints and avoids intermediate sanity checks by maintaining the iteration domain immutable.

Other directive-based tools were proposed to implement high-level loop transformations. Xlanguage [6] uses compiler pragmas for syntactic loop transformations in the C source code. A scripting language POET [23] stores and parameterizes AST-based transformations. Goofi [16] features visual interface for transforming loops in FORTRAN code. None of them benefit from the precise data dependence analysis enabled by the polyhedral model, which is critical when designing complex transformation sequences.

Developer/Compiler Interaction for Optimization To the best of our knowledge, no other work addresses the interaction with a compiler for loop-level optimization. However, several works propose complementary approaches to interact with a compiler for better optimizations. The Paralax infrastructure [20] allows for completing compiler analyses by annotations and uses compiler feedback to suggest them. Larsen et.al. [15] instrument a production compiler to provide feedback as to why an automatic loop parallelization was not possible. Jensen et.al. [12] propose an IDE plug-in incorporating feedback from compiler optimization passes. A tool by Göhringer and Tepelmann [10] gives feedback on the fitness of compute-intensive loops to the polyhedral representation. Prospector [14] uses dynamic profiling to discover loop-level parallelism and provide feedback. Contrary to these tools, our approach avoids manual code modification, relying on high-level interface to express transformations and on polyhedral frameworks for code generation.

9. Conclusion

In this paper, we presented a new way to interact with a compiler. It enables a complete feedback and control over loop-level optimizations which are critical when targeting modern multicore architectures. We introduced a new formalism to translate extended versions of classical syntactic transformations to the state-of-the-art mathematical representation of programs used in compiler polyhedral frameworks. This formalism features a unique capacity at composing complex sequences of transformations to a single abstraction, which allows polyhedral frameworks to check the legality of the whole sequence and to generate the final code automatically. We also presented arguably the first algorithm to retrieve a high-level transformation sequence from the scheduling abstraction. These contributions together virtually open the polyhedral compiler black-box, offering means to polish compiler optimizations or to set user-defined optimizations or to freeze a given optimization strategy.

Ongoing work aims at translating scheduling relations to shorter transformation sequences that would better mimic a manual optimization. We are also using the proposed formalism to design visual program manipulations to help users at designing or refining transformation sequences.

References

[1] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE PACT Intl. Conf. on Parallel Architecture and Compilation Techniques*, France, 2004.

[2] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *CC Intl. Conf. on Compiler Construction*, 2010.

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

[4] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proc. of the 19th Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT'10, 2010.

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

[6] 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.

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

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. Journal of Parallel Programming*, 21(6):389–420, Dec. 1992.

[9] 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. *Int. J. of Parallel Programming*, 34(3):261–317, 2006.

[10] D. Göhringer and J. Tepelmann. An interactive tool based on polly for detection and parallelization of loops. In *Proc. of Intl. Workshop PARMA-DITAM*, Austria, 2014.

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

[12] N. B. Jensen, S. Karlsson, and C. W. Probst. Compiler feedback using continuous dynamic compilation during development. In *Intl. W. on Dynamic Compilation Everywhere*, 2014.

[13] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126.1, 1993.

[14] M. Kim, H. Kim, and C.-K. Luk. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010.

[15] P. Larsen, R. Ladelsky, J. Lidman, S. McKee, S. Karlsson, and A. Zaks. Parallelizing more loops with compiler guided refactoring. In *Parallel Processing (ICPP), Intl. Conf.*, 2012.

[16] R. Müller-Pfefferkorn, W. Nagel, and B. Trenkler. Optimizing cache access: A tool for source-to-source transformations and real-life compiler tests. In *10th Intl. Euro-Par Conf.*, 2004.

[17] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, Canada, 2006.

[18] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08)*, Arizona, 2008.

[19] G. Rudy, M. Murtaza Khan, M. Hall, C. Chen, and J. Chame. A programming language interface to describe transformations and code generation. In *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 136–150, Houston, TX, 2010.

[20] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *PACT'19 IEEE Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 389–400, 2010.

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

[22] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.

[23] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Intl. Parallel and Distributed Processing Symposium IPDPS 2007*.

[24] 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.