A Polyhedral Approach to Ease the Composition of Program Transformations

Albert Cohen [1], Sylvain Girbal [1 2], and Olivier Temam [1]

[1] ALCHEMY Group, INRIA Futurs and LRI, Université Paris-Sud
[2] CEA LIST, Saclay

Abstract. We wish to extend the effectiveness of loop-restructuring compilers by improving the robustness of loop transformations and easing their composition in long sequences. We propose a formal and practical framework for program transformation. Our framework is well suited for iterative optimization techniques searching not only for the appropriate parameters of a given transformation, but for the program transformations themselves, and especially for compositions of program transformations. This framework is based on a unified polyhedral representation of loops and statements, enabling the application of generalized control and data transformations without reference to a syntactic program representation. The key to our framework is to clearly separate the impact of each program transformation on three independent components: the iteration domain, the iteration schedule and the memory access functions. The composition of generalized transformations builds on normalization rules specific to each component of the representation. Our techniques have been implemented on top of Open64/ORC.

1 Introduction

Todays compilers ability to apply and search for compositions of program transformations is limited. Compilers can embed a large array of optimizations, but they are often expressed as a collection of ad-hoc syntactic transformations based on pattern-matching. In addition, control structures are regenerated after each transformation, making it harder to apply the next transformations. Finally, compilers follow a rigid ordering of phases, so that only short and fixed sequences of program transformations can be applied [26]. Current approaches to iterative optimization [1, 11, 8] substitute empirical search strategies to the usual model-driven heuristics, but to not improve the transformation framework itself. Indeed, iterative/adaptive compilers usually choose a rather small set of transformations, e.g., cache tiling, unrolling and array padding, and focus on finding the best possible parameters, e.g., tile size, unroll factor and padding size. O'Boyle et al. [11] and Cooper et al. [8] outlined that the ability to perform long sequences of composed transformations is key to the emergence of practical iterative optimization frameworks. Another recent study [22] confirms that complex compositions of many distinct transformations can bring significant performance benefits.

This article introduces a framework for easily expressing compositions of program transformations, based on the polyhedral representation of programs [10] and on a robust code generation technique [23, 3]. We distinguish three different types of actions performed by program transformations: modification of the iteration domain (loop bounds and strides), modification of the iteration schedule of each statement, and modification of the memory access functions (array subscripts). Current program representations do not clearly reflect this separation, making the implementation and composition of program transformations more complicated. E.g., current implementations of loop fusion incur loop bounds and array subscript modifications that are only byproducts of a schedule transformation (the fused loops are often peeled, increasing code size and making further optimizations more complex). Within our representation, loop fusion is expressed as a schedule transformation with no explicit impact on the iteration domain and memory access. Similarly, a domain transformation like unrolling has no impact on the schedule or memory access functions representations; or a memory access transformation like privatization has no impact on the schedule or domain representations, thus not conflicting with the later application of skewing or unrolling. While our framework is geared toward iterative optimization techniques, it can also facilitate the implementation of statically driven program transformations.

To date, the most thorough application of the polyhedral representation is the Petit dependence analyzer and loop restructuring tool [15] within the Omega project [16]. These tools show that most single loop transformations (both unimodular and non-unimodular) can be modeled as geometric transformations of polyhedra. However, traditional polyhedral representations do not separate the three above-mentioned actions induced by program transformations. Indeed, space-time transformations in the polytope model [10, 25, 15, 18] were aimed at model-based optimizations through operation research algorithms (e.g., linear programming) with no real need for composition sequences. Some polyhedral approaches [15, 9, 19, 12, 24] reproduce or extend classical loop transformations, but ultimately rely on the program syntax for the identification of the loops to operate on. These works require the explicit generation of source code and reconstruction of polyhedra at each transformation step, whether our framework sticks to the polyhedral representation along the whole sequence of transformations. There is a large amount of related works and projects targeting loop-restructuring compilers, see e.g., [7, 13, 5, 14, 16, 17, 15, 21] for representative examples. The associated research report outlines the main comparison points with our approach [6].

This paper does not present performance numbers. The goal is to revisit some theoretical and engineering cornerstones of the design of loop-restructuring compilers. We refer to classical techniques for validation and optimization heuristics [26].

2 Separate Polyhedra for Unified Program Transformations

The polytope model is based on a *semantics*-based representation of loop nests. This representation clearly identifies three separate components: *array access functions* — affine functions describing the mapping of iterations to memory locations — from the *iteration domain* — a geometrical abstraction of loop bounds and strides shaping loop structures — and from the *affine schedule* — another geometrical abstraction of the ordering of iterations and statements. In addition to classical characterization of affine schedules, we also separate the description of iteration ordering of a single statement from inter-statements scheduling.

We assume constant strides and affine bounds for loops; affine array subscripts are hoped for but not mandatory. Within a function body, a *Static Control Part* (SCoP) is a maximal set of consecutive statements without `while` loops, where loop bounds and conditionals only depend on invariants (symbolic constants and surrounding counters) within this set of statements. These invariants are called the *global parameters* of the SCoP. We do not consider procedures, pointers, and inter-SCoP transformations.

The following definitions assume some familiarity with the polytope model: the unfamiliar reader may refer to an associated research report [6]; in addition, Section 3 ends with a short example. Formally, a SCoP is a pair $(\mathcal{S}, \mathbf{i}_{\mathrm{gp}})$, where $\mathcal{S}$ is the set of consecutive *statements* and $\mathbf{i}_{\mathrm{gp}}$ is the vector of *global parameters* (known at run-time). $d_{\mathrm{gp}} = \dim(\mathbf{i}_{\mathrm{gp}})$ denotes the number of global parameters and d^S the depth of statement S, i.e., the number of nested loops enclosing the statement in the SCoP. A statement $S \in \mathcal{S}$ is a quadruple $(\mathcal{D}^S, \mathcal{L}^S, \mathcal{R}^S, \theta^S)$, where $\mathcal{D}^S$ is the d^S-dimensional *iteration domain* of S, $\mathcal{L}^S$ and $\mathcal{R}^S$ denote array references written by S (left-hand side) and read by S (right-hand side) respectively, and θ^S is the *affine schedule* of S, defining the *sequential execution ordering* of iterations of S. [3]

Iteration domains. We denote matrices by capital letters. $\mathcal{D}^S$ is a *convex polyhedron* defined by matrix $\Lambda^S \in \mathcal{M}_{n, d^S + d^S_{\mathrm{lv}} + d_{\mathrm{gp}} + 1}(\mathbb{Z})$ such that

$$\mathbf{i} \in \mathcal{D}^S \iff \exists \mathbf{i}_{\mathrm{lv}}, \Lambda^S \cdot \left(\mathbf{i}, \mathbf{i}_{\mathrm{lv}}, \mathbf{i}_{\mathrm{gp}}, 1\right)^t \geq 0$$

where Λ^S is the matrix defining the domain inequalities; n is the number of inequalities necessary to define the domain (the number of matrix rows, a priori not limited); 1 adds a matrix column to specify the affine component of each domain inequality; and d^S_{lv} is the number of *local variables* required to implement integer division and modulo operations via *affine projection*.

Statements guarded by non-convex conditionals — such as $1 \leq i \leq 3 \vee i \geq 8$ — are split into separate statements with convex domains in the polyhedral representation.

Memory access functions. $\mathcal{L}^S$ and $\mathcal{R}^S$ are *sets* of $(\mathtt{A}, f)$ pairs, where `A` is an array variable and f is the *access function* mapping iterations in $\mathcal{D}^S$ to locations in `A`. The access function f is defined by a matrix $\mathrm{F} \in \mathcal{M}_{\dim(\mathtt{A}), d^S + d^S_{\mathrm{lv}} + d_{\mathrm{gp}} + 1}(\mathbb{Z})$ such that

$$f(\mathbf{i}) = \mathrm{F} \cdot \left(\mathbf{i}, \mathbf{i}_{\mathrm{lv}}, \mathbf{i}_{\mathrm{gp}}, 1\right)^t.$$

Access functions only describe *affine* references; other references are allowed if the dependence analysis framework can manage them [27, 2].

Affine schedules. θ^S is the *affine schedule* of S; it maps iterations in $\mathcal{D}^S$ to multidimensional *time stamps*, i.e., logical execution dates. Multidimensional time stamps are compared through the *lexicographic ordering* over vectors, denoted by $\ll$: iteration $\mathbf{i}$ of S is executed before iteration $\mathbf{i}'$ of S' if and only if $\theta^S(\mathbf{i}) \ll \theta^{S'}(\mathbf{i}')$.

θ^S is defined by a matrix $\Theta^S \in \mathcal{M}_{2d^S+1, d^S + d_{\mathrm{gp}} + 1}(\mathbb{Z})$ such that

$$\theta^S(\mathbf{i}) = \Theta^S \cdot \left(\mathbf{i}, \mathbf{i}_{\mathrm{gp}}, 1\right)^t.$$

---

[3] The term polyhedron will be used in a broad sense to denote a linearly-bounded lattice, i.e., a set of points in a $\mathbb{Z}$ vector space bounded by affine inequalities.

Θ^S does not involve local variables, they would be redundant with the iterators they are related to. Notice the number of rows is $2d^S+1$ and not d^S: to define the relative ordering of statements across iterations at depth k, we need d^S dimensions; to define the relative ordering of statements within each iteration, we need an additional dimension for each depth plus depth 0, hence the $2d^S+1$ dimensions. This encoding was proposed before [10, 15], to model classical transformations into the polytope model.

The schedule matrix is decomposed in a form amenable to transformation composition and scalable code generation; it consists of three sub-matrices: a square *iteration ordering matrix* $\mathrm{A}^S \in \mathcal{M}_{d^S,d^S}(\mathbb{Z})$ operating on iteration vectors, a *statement ordering vector* $\beta^S \in \mathbb{N}^{d^S+1}$, and $\Gamma^S \in \mathcal{M}_{d^S,d_{\mathrm{gp}}+1}(\mathbb{Z})$ called a *parameterization matrix*. The structure of the schedule matrix Θ^S is shown below. $\mathrm{A}^S_{i,j}$ capture the iteration order of S with respect to surrounding loop counters. β^S_i specify the ordering of S among all other statements executed at the same iteration; the first row of Θ^S corresponds to depth 0, the outermost level.[4] $\Gamma^S_{i,j}$ extend the nature of possible transformations, allowing iteration advances and delays by constant or parametric amounts.

$$\Theta^S = \begin{bmatrix} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta^S_0 \\ \mathrm{A}^S_{1,1} & \cdots & \mathrm{A}^S_{1,d^S} & \Gamma^S_{1,1} & \cdots & \Gamma^S_{1,d_{\mathrm{gp}}} & \Gamma^S_{1,d_{\mathrm{gp}}+1} \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta^S_1 \\ \mathrm{A}^S_{2,1} & \cdots & \mathrm{A}^S_{2,d^S} & \Gamma^S_{2,1} & \cdots & \Gamma^S_{2,d_{\mathrm{gp}}} & \Gamma^S_{2,d_{\mathrm{gp}}+1} \\ \vdots & \ddots & \vdots & 0 & \ddots & 0 & \vdots \\ \mathrm{A}^S_{d^S,1} & \cdots & \mathrm{A}^S_{d^S,d^S} & \Gamma^S_{d^S,1} & \cdots & \Gamma^S_{d^S,d_{\mathrm{gp}}} & \Gamma^S_{d^S,d_{\mathrm{gp}}+1} \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta^S_{d^S} \end{bmatrix}.$$

Towards a normalized representation. A given program can have multiple representations, and that, in turn, can limit the application of transformations. E.g., a condition for fusion is that statements must be consecutive; otherwise, if there is a statement in between, one must first decide where to move it (and check dependences). Normalization conditions avoiding these pitfalls are called *invariants*. Besides avoiding useless composition prohibitions, these invariants also serve to avoid matrix parameters overflow.

The schedule density invariant is to ensure that all statements at an identical depth have a consecutive β ordering (no gap). As a side-effect, this invariant also avoids integer overflows on the β parameters, $\beta^S_k > 0 \Rightarrow \exists S' \in \mathcal{S}, \mathrm{pfx}(\beta^S,k) = \mathrm{pfx}(\beta^{S'},k) \wedge \beta^{S'}_k = \beta^S_k - 1$, where $\mathrm{pfx}(\beta^S,k)$ denotes the k first dimensions of vector β^S. The condition states that, for any non-null β parameter at dimension k, there necessarily exists another statement S' with the same k-prefix and the preceding value at dimension k.

The domain parameter invariant avoids redundant inequalities and integer overflows in the domain matrix Λ^S parameters. For that purpose, we impose that the coefficients in a row of Λ^S are always relatively prime.

The sequentiality invariant states that two distinct statements, or two identical statements in distinct iterations, cannot have the same time stamp: $S \neq S' \vee \mathbf{i} \neq \mathbf{i}' \Rightarrow \theta^S(\mathbf{i}) \neq \theta^{S'}(\mathbf{i}')$. A sufficient (though not necessary) condition to enforce that property is the following: $|\det(\mathrm{A}^S)| = 1$ (unimodular) and $S \neq S' \Rightarrow \beta^S \neq \beta^{S'}$.

[4] Notice the first component of β is numbered β_0.

3 Polyhedral Program Transformations

In our framework, program transformations take the form of a set of elementary operations on matrices and vectors describing a SCoP.

We first define elementary operations called *constructors*. Given a vector v and matrix M with $\dim(v)$ columns and at least i rows, $\mathsf{AddRow}(\mathrm{M},i,v)$ inserts a new row at position i in M and fills it with the value of vector v, $\mathsf{RemRow}(\mathrm{M},i)$ does the opposite transformation. $\mathsf{AddCol}(\mathrm{M},j,v)$ and $\mathsf{RemCol}(\mathrm{M},j)$ play similar roles for columns. Moving a statement S forward or backward is a common operation: the constructor $\mathsf{Move}(P,Q,o)$ leaves all statements unchanged except those satisfying

$$\forall S \in \mathcal{S}, P \sqsubseteq \beta^S \wedge (Q \ll \beta^S \vee Q \sqsubseteq \beta^S) : \beta^S_{\dim(P)} \leftarrow \beta^S_{\dim(P)} + o,$$

where $u \sqsubseteq w$ denotes that u is a prefix of v, where P and Q are *statement ordering prefixes* s.t. $P \sqsubseteq Q$ defining respectively the context of the move and marking the initial time-stamp of statements to be moved, and where offset o is the value to be added/subtracted to the component at depth $\dim(P)$ of any statement ordering vector β^S prefixed by P and following Q. If o is positive, $\mathsf{Move}(P,Q,o)$ inserts o free slots before all statements S preceded by the statement ordering prefix Q at the depth of P; respectively, if o is negative, $\mathsf{Move}(P,Q,o)$ deletes $-o$ slots. These constructors make no assumption about the representation invariants and may violate them.

3.1 Composition of Primitives

From the earlier constructors, we define invariant-enforcing transformation *primitives* to serve as building blocks for transformation sequences. Figure 1 lists typical primitives affecting the polyhedral representation of a statement. $\mathbf{1}_k$ denotes the vector filled with zeros but element k set to 1, i.e., $(0,\ldots,0,1,0,\ldots,0)$; likewise, $\mathbf{1}_{i,j}$ denotes the matrix filled with zeros but element (i,j) set to 1.

Like the Move *constructor, primitives do not directly operate on loops or statements, but target a collection of statements and polyhedra whose statement-ordering vectors share a common prefix P. There are no prerequisites on the program representation to the application and composition of primitives.*

We also specified a number of optional *validity prerequisites* that conservatively check for the semantical soundness of the transformation, e.g., there are validity prerequisites to check that no dependence is violated by a unimodular or array contraction transformation. When exploring the space of possible transformation sequences, validity prerequisites avoid wasting time on corrupt transformations.

FUSION and FISSION best illustrate the benefit of designing loop transformations at the abstract semantical level of our unified polyhedral representation. First of all, loop bounds are not an issue since the code generator will handle any overlapping of iteration domains. For the fission primitive, vector (P,o) prefixes all statements concerned by the fission; and parameter b indicates the position where statement delaying should occur. For the fusion primitive, vector $(P,o+1)$ prefixes all statements that should be interleaved with statements prefixed by (P,o). Eventually, notice that fusion followed by fission (with the appropriate value of b) leaves the SCoP unchanged.

UNIMODULAR implements any unimodular transformation, extended to arbitrary iteration domains and loop nesting. U and V denote unimodular matrices.

SHIFT is a kind of source-level hierarchical software pipeline, extended with parametric forward/backward iteration shifts, e.g., to delay a statement by N iterations of one surrounding loop. Matrix M implements the parameterized shift of the affine schedule of a statement. M must have the same dimension as Γ.

RESTRICT constrains the domain with an additional inequality, given in the form of a vector c with the same dimension as a row of matrix Γ.

EXTEND inserts a new intermediate loop level at depth ℓ, initially restricted to a single iteration. This new iterator will be used in following code transformations.

ADDLVDOM and ADDLVACC insert a fresh local variable in the domain and and access functions, respectively. This local variable is typically used by RESTRICT.

PRIVATIZE and CONTRACT implement basic forms of array privatization and contraction, respectively, considering dimension ℓ of the array. Privatization needs an additional parameter s, the size of the additional dimension; s is required to update the array declaration (it cannot be inferred in general, some references may not be affine). These primitives are simple examples updating the data layout and array access functions.

Although this table is not complete, it demonstrates the expressiveness of the unified representation through classical *control and data* transformations.

This table is not complete (e.g., it lacks index-set splitting and data-layout transformations), but it demonstrates the expressiveness of the unified representation.

Syntax	Effect
UNIMODULAR $(P, \mathrm{U}, \mathrm{V})$	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \mathrm{A}^S \leftarrow \mathrm{U}.\mathrm{A}^S.\mathrm{V}$
SHIFT (P, M)	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \Gamma^S \leftarrow \Gamma^S + \mathrm{M}$
RESTRICT (P, c)	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \Lambda^S \leftarrow \mathsf{AddRow}\big(\Lambda^S, 0, c/\gcd(c_1, \ldots, c_{d^S + d^S_{\mathrm{lv}} + d_{\mathrm{gp}} + 1})\big)$
EXTEND (P, ℓ, c)	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \begin{cases} d^S \leftarrow d^S + 1;\ \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S, c, 0);\ \beta^S \leftarrow \mathsf{AddRow}(\beta^S, \ell, 0); \\ \mathrm{A}^S \leftarrow \mathsf{AddRow}(\mathsf{AddCol}(\mathrm{A}^S, c, 0), \ell, \mathbf{1}_\ell);\ \Gamma^S \leftarrow \mathsf{AddRow}(\Gamma^S, \ell, 0); \\ \forall (\mathtt{A}, \mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F}, \ell, 0) \end{cases}$
ADDLVDOM (P)	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, d^S_{\mathrm{lv}} \leftarrow d^S_{\mathrm{lv}} + 1;\ \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S, d^S + 1, 0)$
ADDLVACC $(P, \mathtt{A})$	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \forall (\mathtt{A}, \mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{AddCol}(\mathrm{F}, d^S + 1, 0)$
PRIVATIZE $(\mathtt{A}, \ell)$	$\dim(\mathtt{A}) \leftarrow \dim(\mathtt{A}) + 1,\ \forall S \in \mathcal{S}, \forall (\mathtt{A}, \mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F}, \ell, \mathbf{1}_\ell)$
CONTRACT $(\mathtt{A}, \ell)$	$\dim(\mathtt{A}) \leftarrow \dim(\mathtt{A}) - 1;\ \forall S \in \mathcal{S}, \forall (\mathtt{A}, \mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{RemRow}(\mathrm{F}, \ell)$
FUSION (P, o)	$b = \max\{\beta^S_{\dim(P)+1} \mid (P, o) \sqsubseteq \beta^S\} + 1$ $\mathsf{Move}((P, o+1), (P, o+1), b);\ \mathsf{Move}(P, (P, o+1), -1)$
FISSION (P, o, b)	$\mathsf{Move}(P, (P, o, b), 1);\ \mathsf{Move}((P, o+1), (P, o+1), -b)$

Fig. 1. Some classical transformation primitives

Primitives operate on program representation while maintaining the structure of the polyhedral components (the invariants).

Despite their familiar names, the primitives' practical outcome on the program representation is widely extended compared to their syntactic counterparts. Indeed, transformation primitives like fusion or interchange apply to sets of statements that may be scattered and duplicated at many different locations in the generated code. In addition, these transformations are not proper *loop* transformations anymore, since they apply to sets of statement iterations that may have completely different domains and relative iteration schedules. For example, one may interchange the loops surrounding one statement in a loop body without modifying the schedule of other statements, and without distributing the loop first. Another example is the fusion of two loops with different domains without peeling any iteration.

Previous encodings of classical transformations in a polyhedral setting — most significantly [25] and [15] — use Presburger arithmetic as an expressive *operating* tool for implementing and validating transformations. In addition to operating on polytopes, our work *generalizes* loop transformations to more abstract *polyhedral domain* transformations, without explicitly relying on a nested loop structure with known bounds and array subscripts to define the transformation.

Instead of anchoring loop transformations on a syntactic program form, limitting ourselves to what can be expressed with an imperative semantics, we define higher level transformations on the polyhedral representation itself, abstracting away the overhead (versioning, duplication) and constraints of the code generation process (translation to an imperative semantics).

Syntax	Effect	Comments
$\textsc{Interchange}\,(P,o)$	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \begin{cases} \mathrm{V} = \mathbf{I}_{d^S} - \mathbf{1}_{o,o} - \mathbf{1}_{o+1,o+1} + \mathbf{1}_{o,o+1} + \mathbf{1}_{o+1,o}; \\ \textsc{Unimodular}\,(\beta^S, \mathbf{I}_{d^S}, \mathrm{V}) \end{cases}$	swap rows o and $o+1$
$\textsc{Skew}\,(P,\ell,c,s)$	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \mathrm{V} = \mathbf{I}_{d^S} + s \cdot \mathbf{1}_{\ell,c};\ \textsc{Unimodular}\,(\beta^S, \mathbf{I}_{d^S}, \mathrm{V})$	add the skew factor
$\textsc{Reverse}\,(P,o)$	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \mathrm{V} = \mathbf{I}_{d^S} - 2 \cdot \mathbf{1}_{o,o};\ \textsc{Unimodular}\,(\beta^S, \mathbf{I}_{d^S}, \mathrm{V})$	put a -1 in (o,o)
$\textsc{StripMine}\,(P,k)$	$\forall S \in \mathcal{S} \mid P \sqsubseteq \beta^S, \begin{cases} \ell = \dim(P);\ c = \dim(P); \\ \textsc{Extend}\,(\beta^S, \ell, c); \\ \textsc{AddLVDom}\,(\beta^S); \\ p = d^S + 1;\ u = d^S + d^S_{\mathrm{lv}} + d_{\mathrm{gp}} + 1; \\ \textsc{Restrict}\,(\beta^S, \mathrm{A}^S_{\ell+1} - \mathbf{1}_c); \\ \textsc{Restrict}\,(\beta^S, -\mathrm{A}^S_{\ell+1} + \mathbf{1}_c + (k-1)\mathbf{1}_u); \\ \textsc{Restrict}\,(\beta^S, \mathbf{1}_c - k \cdot \mathbf{1}_p); \\ \textsc{Restrict}\,(\beta^S, k \cdot \mathbf{1}_p - \mathbf{1}_c) \end{cases}$	insert interm. loop insert local var. local var. and const. $\mathbf{i}_c \le \mathbf{i}_{c+1}$ $\mathbf{i}_{o+1} \le \mathbf{i}_o + k - 1$ $k \times p \le ii$ $ii \le k \times p$
$\textsc{Tile}\,(P,o,k)$	$\forall S \in \mathcal{S} \mid (P,o) \sqsubseteq \beta^S, \begin{cases} \textsc{StripMine}\,(P,k); \\ \textsc{StripMine}\,((P,o),k); \\ \textsc{Interchange}\,((P,0),\dim(P)) \end{cases}$	strip outer loop strip inner loop interchange

Fig. 2. Composition of transformation primitives

Naturally, this higher-level framework is beneficial for transformation composition. Figure 2 composes primitives into typical transformations. INTERCHANGE swaps the roles of $\mathbf{i}_o$ and $\mathbf{i}_{o+1}$ in the schedule of the matching statements; it is a fine-grain extension of the classical interchange making no assumption about the shape of the iteration domain. SKEW and REVERSE define two well known unimodular transformations, with respectively the skew factor s with it's coordinates (ℓ, c), and the depth o of the iterator to be reversed. STRIPMINE introduces a new iterator to unroll k times the schedule and iteration domain of all statements at the depth of P (where k is a *statically known integer*). This transformation is a complex sequence of primitives, see Figure 2. TILE extends the classical loop tiling at of the two nested loops at the depth of P, using $k \times k$ blocks, with arbitrary nesting and iteration domains. Tiling and strip-mining always operate on *time* dimensions; it is possible to tile the surrounding time dimensions of any collection of statements with unrelated iteration domains and schedules.

Other properties of our framework include confluence and commutativity, when operating on distinct components of the representation. Further exploration of these properties is under way, in an attempt to improve the structure of the transformation space for iterative optimization purposes.

3.2 Composition Example

Code complexity after loop transformations is mainly due to control optimizations (hoisting of conditionals, unrolling) which do not affect the complexity of our representation. The main asset of our framework is to hide the code complexity along the sequence. At intermediate steps, the complexity of the code representation within our framework remains fairly low, i.e., it only depends linearly on the number of original statements and statement-insertions. Using syntactic program transformations, the code complexity may increase at each intermediate step, sometimes even preventing further optimizations [6]. Notice that transformation with an intrinsic syntactical behavior like loop unrolling may also benefit from a polyhedral representation: one may strip-mine the loop instead, delaying the proper unrolling to the code generation step, and separate transformations may still be applied to each virtually unrolled iteration.

Let us compare our framework with syntactic transformations, studying the evolution of the representation along a sequence of transformations. The example in Figure 3 performs two matrix-vector multiplications, yielding $D =^t BEC$ (typical of quadratic form computations), where arrays `B` and `E` store $M \times N$ rectangular matrices.

We apply a sequence of three transformations to this program. In Figure 4, we interchange the loops in the first nest to optimize spatial locality on `B`. In Figure 5, we fuse the outer loops to improve temporal locality on `A`. Figure 6 shows part of the resulting code after advancing assignments to `A` by 4 iterations, in order to cover the latency of floating-point multiplications. This sequence corresponds to the following composition of primitives:

$$\textsc{Interchange}\big(\beta^{S_1},1\big);\ \textsc{Fusion}\big((),0\big);\ \textsc{Shift}\big(\beta^{S_1},\left[\begin{smallmatrix}0&0&0\\0&0&-4\end{smallmatrix}\right]\big).$$

Based on the final polyhedral representation, the code generation phase will generate control-optimized code quite similar to the hand-optimized fragment in Figure 6, without redundant guards or dead iterations.

4 Implementation

This framework uses the PolyLib [20] and CLooG — a robust new code generator [3]. It is implemented as a plug-in for Open64/ORC.[5] The availability of CLooG is a major reason for making polyhedral approaches applicable to real codes. Our tool converts the WHIRL, the intermediate representation of Open64, to an augmented polyhedral representation with maps to the symbol table and syntax tree, and regenerates WHIRL from this representation [4]. 12 SpecFP 2000 and PerfectClub benchmarks were run through the whole source-to-polyhedra-to-source conversion cycle (without loop transformations); it takes from one second to two minutes on a typical Pentium III, see [4]

[5] Publicly available at `http://www-rocq.inria.fr/a3/wrap-it`.

```
       do i = 1, M
         do j = 1, N
(S1)       A(i) += B(i,j)*C(j)
       do k = 1, M
         do l = 1, N
(S2)       D(k) += E(l,k)*A(l)
```

$$\mathcal{D}^{S_1} = \{i,j \mid 1 \le i \le M \,;\, 1 \le j \le N\}, \mathrm{A}^{S_1} = \begin{bmatrix}1&0\\0&1\end{bmatrix}, \Gamma^{S_1} = \begin{bmatrix}0&0&0\\0&0&0\end{bmatrix}, \beta^{S_1} = \begin{bmatrix}0\\0\\0\end{bmatrix}$$

$$\mathcal{D}^{S_2} = \{k,l \mid 1 \le k \le M \,;\, 1 \le l \le N\}, \mathrm{A}^{S_2} = \begin{bmatrix}1&0\\0&1\end{bmatrix}, \Gamma^{S_2} = \begin{bmatrix}0&0&0\\0&0&0\end{bmatrix}, \beta^{S_2} = \begin{bmatrix}1\\0\\0\end{bmatrix}$$

$$\mathcal{L}^{S_1} = \left(\mathtt{A}, \begin{bmatrix}1&0&0&0&0\end{bmatrix}\right), \mathcal{L}^{S_2} = \left(\mathtt{B}, \begin{bmatrix}1&0&0&0&0\end{bmatrix}\right)$$

Fig. 3. Original code and its representation

```
do j = 1, N
  do i = 1, M
    A(i) += B(i,j)*C(j)
do k = 1, M
  do l = 1, N
    D(k) += E(l,k)*A(l)
```

Changes to the polyhedral representation: $\mathrm{A}^{S_1} = \begin{bmatrix}0&1\\1&0\end{bmatrix}$

Fig. 4. After interchange

```
MN = min(M,N)
do x = 1, MN
  do y = 1, MN
    A(y) += B(y,x)*C(x)
    D(x) += E(y,x)*A(y)
  do y = MN+1, M
    A(y) += B(y,x)*C(x)
    D(y) += E(y,x)*A(y)
do x = MN+1, N
  do y = 1, MN
    A(y) += B(y,x)*C(x)
    D(y) += E(y,x)*A(y)
```

Changes to the polyhedral representation: $\beta^{S_2} = \begin{bmatrix}0\\0\\1\end{bmatrix}$

Fig. 5. After fusion

```
MN = min(M-4,N)
do x = 1, MN
  A(1) += B(1,x)*C(x)
  A(2) += B(2,x)*C(x)
  A(3) += B(3,x)*C(x)
  A(4) += B(4,x)*C(x)
  do y = 1, MN
    A(y+4) += B(y+4,x)*C(x)
    D(x) += E(y,x)*A(y)
  D(x) += E(MN-3,x)*A(MN-3)
  D(x) += E(MN-2,x)*A(MN-2)
  D(x) += E(MN-1,x)*A(MN-1)
  D(x) += E(MN,x)*A(MN)
  do y = MN+1, M-4
    A(y+4) += B(y+4,x)*C(x)
    D(y) += E(y,x)*A(y)
  D(M-3) += E(MN-3,x)*A(MN-3)
  D(M-2) += E(MN-2,x)*A(MN-2)
  D(M-1) += E(MN-1,x)*A(MN-1)
  D(M) += E(MN,x)*A(MN)
do x = MN+1, N
  do y = 1, M-4
...
```

Changes to the polyhedral representation: $\Gamma^{S_1} = \begin{bmatrix}0&0&0\\0&0&-4\end{bmatrix}$

Fig. 6. After four shifts (partial code fragment)

and [3] for details. The software component in charge of the polyhedral transformations is driven through a script language, with a specific syntax to define primitives and to compose new transformations. The syntax is based on C++ with overloaded operators for vector and matrices. From the script, the tool generates the actual source code of the related transformations: it effectively generates a class for each transformation with its own methods for prerequisite checking and application.

5 Conclusion

We presented a polyhedral framework that enables the composition of long sequences of program transformations. Coupled with a robust code generator, this method avoids the typical code complexity explosion of long compositions of program transformations; these techniques have been implemented in the Open64/ORC compiler. While revisiting the design of a polyhedral program representation and the definition of transformation primitives, we have shown the advantages of using such advanced techniques in the software engineering of a loop-restructuring compiler. In particular, we have decoupled the polyhedral representation, transformation and code generation techniques from their historical applications to automatic affine scheduling and mapping (automatic parallelization and model-based optimization). More practically, the ability to perform numerous compositions of program transformations is key to the extension of iterative optimizations to finding the appropriate program transformations instead of just the appropriate program transformation parameters.

Acknowledgments This work is supported by the "COP" RNTL grant from the French Ministry of Research. We are very thankful to Cedric Bastoul and Saurabh Sharma, whose algorithms, tools and support have been critical for the design and implementation of our framework.

References

1. M. Barreteau, F. Bodin, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. R. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, T. Kisuki, P. M. W. Knijnenburg, P. van der Mark, A. Nisbet, M. F. P. O'Boyle, E. Rohou, A. Seznec, E. Stöhr, M. Treffers, and H. A. G. Wijshoff. Oceans - optimising compilers for embedded applications. In *Euro-Par'99*, pages 1171–1775, Aug. 1999.
2. D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. of Parallel and Distributed Computing*, 40:210–226, 1997.
3. C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubjana, Slovenia, Oct. 2003.
4. C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, College Station, Texas, Oct. 2003.
5. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.
6. A. Cohen, S. Girbal, and O. Temam. Facilitating the exploration of compositions of program transformations. Research report 5114, INRIA Futurs, France, Feb. 2004.
7. K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.
8. K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.
9. A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, Boston, 2000.
10. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.
11. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In 11^{th} *Workshop on Languages and Compilers for Parallel Computing*, LNCS, Washington DC, July 2002. Springer-Verlag.
12. A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware design methodology with the Alpha language. In *FDL'01*, Lyon, France, Sept. 2001.
13. M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.
14. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Int. Conf. on Supercomputing (ICS'2)*, Cologne, Germany, June 1991.
15. W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.
16. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symp. on the frontiers of massively parallel computation*, McLean, 1995.
17. I. Kodukula and K. Pingali. Transformations for imperfectly nested loops. In *Supercomputing (SC'96)*, Jan. 1996.
18. A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In 24^{th} *ACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.
19. A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'01*, pages 102–112, 2001.
20. V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997. http://icps.u-strasbg.fr/PolyLib.
21. M. O'Boyle. MARS: a distributed memory approach to shared memory compilation. In *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Pittsburgh, May 1998. Springer-Verlag.
22. D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance? matrix-multiply revisited. In *SuperComputing'02*, Baltimore, Maryland, Nov. 2002.
23. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, Oct. 2000.
24. R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. Technical report, Hewlett-Packard, May 2000.
25. M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Aug. 1992. Published as CSL-TR-92-538.
26. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
27. D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.