Putting Polyhedral Loop Transformations to Work

Cédric Bastoul[1,3], Albert Cohen[1], Sylvain Girbal[1,2,4], Saurabh Sharma[1], and Olivier Temam[2]

[1] A3 group, INRIA Rocquencourt
[2] LRI, Paris South University
[3] PR*i*SM, University of Versailles
[4] LIST, CEA Saclay

Abstract. We seek to extend the scope and efficiency of iterative compilation techniques by searching not only for program transformation parameters but for the most appropriate transformations themselves. For that purpose, we need a generic way to express program transformations and compositions of transformations. In this article, we introduce a framework for the polyhedral representation of a wide range of transformations in a unified way. We also show that it is possible to generate efficient code after the application of polyhedral program transformations. Finally, we demonstrate an implementation of the polyhedral representation and code generation techniques in the Open64/ORC compiler.

1 Introduction

Optimizing and parallelizing compilers face a tough challenge. Due to their impact on productivity and portability, programmers of high-performance applications want compilers to automatically produce quality code on a wide range of architectures. Simultaneously, Moore's law indirectly urges the architects to build complex architectures with deeper pipelines and (non uniform) memory hierarchies, wider general-purpose and embedded cores with clustered units and speculative structures. Static cost models have a hard time coping with rapidly increasing architecture complexity. Recent research works on iterative and feedback-directed optimizations [17] suggest that practical approaches based on dynamic information can better harness complex architectures.

Current approaches to iterative optimizations usually choose a rather small set of program transformations, e.g., cache tiling and array padding, and focus on finding the best possible transformation parameters, e.g., tile size and padding size [17] using parameter search space techniques. However, a recent comparative study of model-based vs. empirical optimizations [22] stresses that many motivations for iterative, feedback-directed or dynamic optimizations are irrelevant when the proper transformations are not available. We want to extend the scope and efficiency of iterative compilation techniques by making the program transformation itself one of the parameters. Moreover, we want to search for composition of program transformations and not only single program transformations. For that purpose, we need a generic method for expressing program transformations and composition of those.

This article introduces a unified framework for the implemention and composition of generic program transformations. This framework relies on a polyhedral representation of loops and loop transformations. By separating the iteration domains from the statement and iteration schedules, and by enabling per-statement transformations, this representation avoids many of the limitations of iteration-based program transformations, widens the set of possible transformations and enables parameterization. Few invariants constrain the search space and our *non-syntactic representation* imposes no ordering and compatibility constraints. In addition, statements are *named independenty from their location and surrounding control structures*: this greatly simplifies the practical description of transformation sequences. We beleive this generic expression is appropriate for systematic search space techniques.

The corresponding search techniques and performance evaluations are out of the scope of this work and will be investigated in a follow-up article. This work presents the principles of our unified framework and the first part of its implementation. Also, since polyhedral transformation techniques can better accommodate complex control structures than traditional loop-based transformations, we start with an empirical study of control structures within a set of benchmarks. The four key aspects of our research work are: (1) empirically evaluating the scope of polyhedral program transformations, (2) defining a practical transformation environment based on a polyhedral representation, (3) showing that it is possible to generate efficient code from a polyhedral transformation, (4) implementing the polyhedral representation and code generation technique in a real compiler, Open64/ORC [18], with applications to real benchmarks.

Eventually, our framework operates at an abstract semantical level to hide the details of control structures, rather than on a syntax tree. It allows per-statement and extended transformations that make few assumptions about control structures and loop bounds. Consequently, while our framework is initially geared toward iterative optimization techniques, it can also facilitate the implementation of statically driven program transformations in a traditional optimizing compiler.

The paper is organized as follows. We present the empirical analysis of static control structures in Section 2 and discuss their significance in typical benchmarks. The unified transformation model is described in Section 3. Section 4 presents the code generation techniques used after polyhedral transformations. Finally, implementation in Open64/ORC is described in Section 5.

2 Static Control Parts

Let us start with some related works. Since we did not directly contribute to the driving of optimizations and parallelization techniques, we will not compare with the vast literature in the field of model-based and empirical optimization.

Well-known loop restructuring compilers proposed unified models and intermediate representations for loop transformations, but none of them addressed the general composition and parameterization problem of polyhedral techniques. ParaScope [6] is both a dependence-based framework and an interactive source-to-source compiler for Fortran; it implements classical loop transformations. SUIF [11] was designed as an intermediate representation and framework for automatic loop restructuring; it quickly became a standard platform for implementing virtually any optimization prototype, with multiple front-ends, machine-dependent back-ends and variants. Polaris [4] is an automatic parallelizing compiler for Fortran; it features a rich sequence of analyzes and loop transformations applicable to real benchmarks. These three projects are based on a syntax-tree representation, ad-hoc dependence models and implement polynomial algorithms. PIPS [12] is probably the most complete loop restructuring compiler, implementing polyhedral analyses and transformations (including affine scheduling) and interprocedural analyses (array regions, alias). PIPS uses an expressive intermediate representation, a syntax-tree with polyhedral annotations.

Within the Omega project [14], the Petit dependence analyzer and loop restructuring tool [13] is much closer to our work: it provides a unified polyhedral framework (space-time mappings) for iteration reordering only, and it shares our emphasis on per-statement transformations. It is intended as a research tool for small kernels only. The MARS compiler [16] is also very close to our work: its polyhedral representation allows to unify several loop transformations to ease the application of long transformation sequences. Its successes in iterative optimization [17] makes it the main comparison point and motivation for our work, although MARS lacks the expressivity of the affine schedules we use in our unified model.

Two codesign projects have a lot in common with our semi-automatic optimization project. MMAlpha [10] is a domain-specific single assignment language for systolic array computations, a polyhedral transformation framework, and a high-level circuit synthesis tool. The interactive and semi-automatic approach to polyhedral transformations were introduced by MMAlpha. The PICO project [20] is a more pragmatic approach to codesign, restricting the application domain to loop nests with uniform dependences and aiming at the selection and coordination of existing functional units to generate an application-specific VLIW processor. Both tools only target small kernels.

2.1 Decomposition into Static Control Parts

In the following, loops are normalized and split in two categories: loops from 0 to some bound expression with an integer stride, called `do` loops; other kinds of loops, referred to as `while` loops. Early phases of the Open64 compiler perform most of this normalization, along with closed form substitution of induction variables. Notice some Fortran and C `while` loops may be normalized to `do` loops when bound and stride can be discovered statically.

The following definition is a slight extension of *static control* nests [8]. Within a function body, a *static control part* (SCoP) is a maximal set of consecutive statements without `while` loops, where loop bounds and conditionals may only depend on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters: they are called the *global parameters* of the SCoP, as well as any invariant appearing in some array subscript within the SCoP. A static control part is called *rich* when it holds at least one non-empty loop; *rich* SCoPs are the natural candidates for polyhedral loop transformations. An example is shown in Figure 1. We will only consider *rich* SCoPs in the following.

As such, a SCoP may hold arbitrary memory accesses and function calls; a SCoP is thus larger than a static control loop nest [8]. Interprocedural alias and array region analysis would be useful for precise dependence analysis. Nevertheless, our semi-automatic framework copes with crude dependence information in authorizing the expert user to override static analysis when applying transformations.

```
                         SCoP decomposition
do i=1, 3
.................................................................
| S1                        | SCoP 1, one statement, non rich
.................................................................
| do j=1, i*i
.................................................................
| | S2                      | SCoP 2, three statements, rich
| | do k=0, j               |    parameters: i,j
| | | if (j .ge. 2) then    |    iterators: k
| | | | S3                  |
| | | S4                    |
.................................................................
| do p = 0, 6               | SCoP 3, two statements, rich
| | S5                      |    iterators: p
| | S6                      |
```

Fig. 1. Example of decomposition into static control parts

2.2 Automatic Discovery of SCoPs

SCoP extraction is greatly simplified when implemented within a modern compiler infrastructure such as Open64/ORC. Previous phases include function inlining, constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, and induction variable substitution, along with language-specific preprocessing: pointer arithmetic is replaced by arrays, pointer analysis information is available (but not yet used in our tool), etc. The algorithm for SCoP extraction is detailed in [2]; it outputs a list of SCoPs associated with any function in the syntax tree. Our implementation in Open64 is discussed in Section 5.

2.3 Significance Within Real Applications

Thanks to an implementation of the previous algorithm into Open64, we studied the applicability of our polyhedral framework to several benchmarks.

Figure 2 summarizes the results for the SpecFP 2000 and PerfectClub benchmarks handled by our tool (single-file programs only, at the time being). Construction of the polyhedral representation takes much less time than the preliminary analyses performed by Open64/ORC. All codes are in Fortran77, except art and quake in C, and lucas in Fortran90. The first column shows the number of functions (inlining was not applied in these experiments). The next two columns count the number of SCoPs with at least one global parameter and enclosing at least one conditional, respectively; the first one advocates for parametric analysis and transformation techniques; the second one shows the need for techniques that handle static-control conditionals. The next two columns in the "Statements" section show that SCoPs cover a large majority of statements (many statements are enclosed in affine loops). The last two columns in the "Array References" section are very promising for dependence analysis: most subscripts are affine except for lucas and mg3d (the rate is over 99% in 7 benchmarks), but approximate array dependence analyses will be required for a good coverage of the 5 others. In accordance with earlier results using Polaris [7], the coverage of regular loop nests is strongly influenced by the quality of loop normalization and induction variable detection.

		SCoPs			Statements		Array References	
	Functions	All	Parametric	Affine `ifs`	All	in SCoPs	All	Affine
applu	16	19	15	1	757	633	1245	100%
apsi	97	80	80	25	2192	1839	977	78%
art	26	28	27	4	499	343	52	100%
lucas	4	4	4	2	2070	2050	411	40%
mgrid	12	12	12	2	369	369	176	99%
quake	27	20	14	4	639	489	218	100%
swim	6	6	6	1	123	123	192	100%
adm	97	80	80	25	2260	1899	147	95%
dyfesm	78	75	70	3	1497	1280	507	99%
mdg	16	17	17	5	530	464	355	84%
mg3d	28	39	39	6	1442	1242	1274	19%
qcd	35	30	23	6	819	641	943	100%

Fig. 2. Coverage of static control parts in high-performance applications

Our tool also gathers detailed statistics about the number of parameters and statements per SCoP, and about statement depth (within a SCoP, not counting non-static enclosing loops). Figure 3 shows that almost all SCoPs are smaller than 100 statements, with a few exceptions, and that loop depth is rarely greater than 3. Moreover, deep loops also tend to be very small, except for applu, adm and mg3d which contain depth-3 loop nests with tenths of statements. This means that most polyhedral analysis and transformations will succeed and require reasonable resources. It also gives an estimate of the scalability required for worst-case exponential algorithms, like the code generation phase to convert the polyhedral representation back to source code.

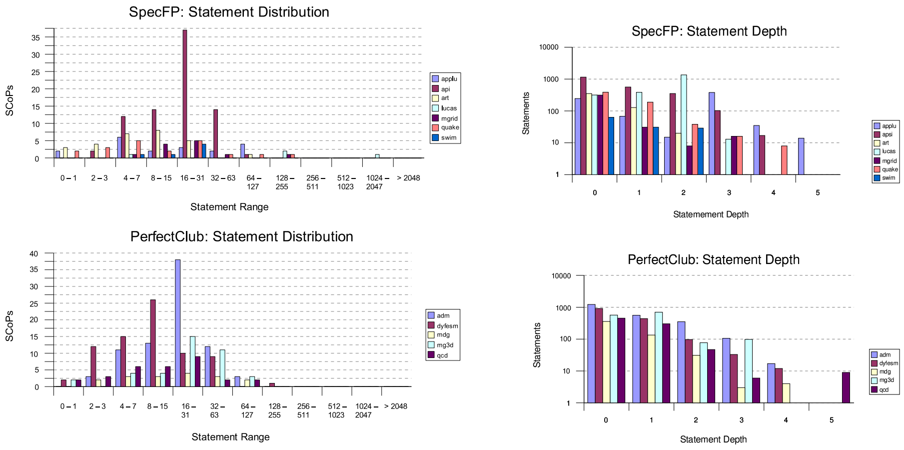


Fig. 3. Distribution of statement depths and SCoP size

3 Unified Polyhedral Representation

In this section, we define the principles of polyhedral program transformations. The term *polyhedron* will be used in a broad sense to denote a *convex set of points in a lattice* (also called $\mathbb{Z}$-polyhedron or lattice-polyhedron), i.e., a set of points in a $\mathbb{Z}$ vector space bounded by affine inequalities.

Let us now introduce the representation of a SCoP and its elementary transformations. A static control part within the syntax tree is a pair $(\mathcal{S}, \mathbf{i}_{\mathrm{gp}})$, where $\mathcal{S}$ is the set of consecutive *statements* — in their polyhedral representation — and $\mathbf{i}_{\mathrm{gp}}$ is the vector of *global parameters* of the SCoP. Vector $\mathbf{i}_{\mathrm{gp}}$ is constant for the SCoP but statically unknown; yet its value is known at runtime, when entering the SCoP. $d_{\mathrm{gp}} = \dim(\mathbf{i}_{\mathrm{gp}})$ denotes the number of global parameters.

We will use a few specific linear algebra notations: matrices are always denoted by capital letters, vectors and functions in vector spaces are not; $\mathrm{pfx}(v,n)$ returns a length-n prefix of v, i.e., the vector built from the n first components of v; $u \sqsubseteq w$ is equivalent to u being a prefix of v; 1_k denotes the k-th unit vector in a reference base $(1_1, \ldots, 1_d)$ of a d-dimensional space, i.e., $(0, \ldots, 0, 1, 0, \ldots, 0)$; likewise, $1_{i,j}$ denotes the matrix filled with zeros but element (i, j) set to 1.

A SCoP may also be decorated with static properties such as array dependences or regions, but this work does not address static analysis.

3.1 Domains, Schedules and Access Functions

The *depth* d^S of a statement S is the number of nested loops enclosing S 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 $\mathcal{R}^S$ are sets of polyhedral representations of *array references*, and θ^S is the *affine schedule* of S, defining the *sequential execution ordering* of iterations of S. To represent arbitrary lattice polyhedra, each statement is provided with a number d^S_{lp} of *local parameters* to implement integer division and modulo operations via *affine projection*: e.g., the set of even values of i is described by means of a local parameter p — existentially quantified — and equation $i = 2p$. Let us describe these concepts in more detail and give some examples.

$\mathcal{D}^S$ is a *convex polyhedron* defined by matrix $\Lambda^S \in \mathcal{M}_{n, d^S + d^S_{\mathrm{lp}} + d_{\mathrm{gp}} + 1}(\mathbb{Z})$ such that

$$\mathbf{i} \in \mathcal{D}^S \iff \exists \mathbf{i}_{\mathrm{lp}}, \Lambda^S \big(\mathbf{i}, \mathbf{i}_{\mathrm{lp}}, \mathbf{i}_{\mathrm{gp}}, 1\big)^t \geq 0.$$

Notice the last matrix column is always multiplied by the constant 1; it corresponds to the *homogeneous coordinate* encoding of *affine* inequalities into *linear* form. The number n of constraints in Λ^S is not limited. Statements guarded by non-convex conditionals — such as $0 \leq i \leq 3 \vee i \geq 8$ — are separated into convex domains in the polyhedral representation. Figure 4 shows an example that illustrates these definitions.

The domains of the five statements are $\mathcal{D}^{S_1} = \{i \mid 1 \leq i \leq N\}$, $\mathcal{D}^{S_2} = \{(i, j) \mid 1 \leq i \leq N, 1 \leq j \leq M\}$, $\mathcal{D}^{S_3} = \{()\}$ (the zero-dimensional vector), $\mathcal{D}^{S_4} = \mathcal{D}^{S_5} = \{k \mid 3 \leq$

```
                       Running example
 do i = 1, N
  | A(i) = 0                                    (S1)
  | do j=1, M
  |  | A(i) = A(i) + B(i, 2*i+j-N-1)            (S2)
 D[0] = 1                                       (S3)
 do k = 3, N, 2
  | D(k) = 2*D(k-2)                             (S4)
  | E(k) = -A(k);                               (S5)
```

Fig. 4. Running example

$k \le N \wedge \exists p, k = 3 + 2p\}$. E.g., the Λ-matrices for statements S_2 and S_4 are

$$\Lambda^{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 \end{bmatrix} \text{ w/ } \left| \begin{array}{l} \mathbf{i} = (i,j) \\ \mathbf{i}_{\text{lp}} = () \\ \mathbf{i}_{\text{gp}} = (N,M) \end{array} \right. \quad \Lambda^{S_4} = \begin{bmatrix} 1 & 0 & 0 & 0 & -3 \\ -1 & 0 & 1 & 0 & 0 \\ 1 & -2 & 0 & 0 & -3 \\ -1 & 2 & 0 & 0 & 3 \end{bmatrix} \text{ w/ } \left| \begin{array}{l} \mathbf{i} = (k) \\ \mathbf{i}_{\text{lp}} = (p) \\ \mathbf{i}_{\text{gp}} = (N,M) \end{array} \right.$$

$\mathcal{L}^S$ and $\mathcal{R}^S$ describe array references written by S (left-hand side) or read by S (right-hand side), respectively; it is a set of pairs $(\texttt{A}, f)$ 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 $\mathrm{F} \in \mathcal{M}_{\dim(\texttt{A}), d^S + d^S_{\text{lp}} + d_{\text{gp}} + 1}(\mathbb{Z})$ such that

$$f(\mathbf{i}) = \mathrm{F}\big(\mathbf{i}, \mathbf{i}_{\text{lp}}, \mathbf{i}_{\text{gp}}, 1\big)^t.$$

E.g., $\mathcal{L}^{S_2} = \big\{ \big(\texttt{A}, (i)\big) \big\}$ and $\mathcal{R}^{S_2} = \big\{ \big(\texttt{A}, (i)\big), \big(\texttt{B}, (i, 2*i + j - N - 1)^t\big) \big\}$, stored as

$$\begin{array}{r} \mathcal{L}^{S_2} : \left\{ \left(\texttt{A}, \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \right) \right\} \\ \mathcal{R}^{S_2} : \left\{ \left(\texttt{A}, \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \right), \left(\texttt{B}, \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & -1 & 0 & -1 \end{bmatrix} \right) \right\} \end{array} \text{ with } \left| \begin{array}{l} \mathbf{i} = (i,j) \\ \mathbf{i}_{\text{lp}} = () \\ \mathbf{i}_{\text{gp}} = (N,M) \end{array} \right.$$

θ^S is the *affine schedule* of S; it maps iterations in $\mathcal{D}^S$ to *time-stamps* (i.e., logical execution dates) in $2d^S + 1$-dimensional time [8]. 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 $\theta^S(\mathbf{i}) \ll \theta^{S'}(\mathbf{i}')$.

To facilitate code generation and to schedule iterations and statements independently, we need $2d^S + 1$ time dimensions instead of d^S (the minimum for a sequential schedule). This encoding was first proposed by Feautrier [8] and used extensively by Kelly and Pugh [13]: dimension $2k$ encodes the relative ordering of statements at depth k and dimension $2k - 1$ encodes the ordering of iterations in loops at depth k.

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

$$\theta^S(\mathbf{i}) = \Theta^S\big(\mathbf{i}, \mathbf{i}_{\text{gp}}, 1\big)^t.$$

Notice Θ^S does not involve local parameters, since lattice polyhedra do not increase the expressivity of sequential schedules. The schedules for the previous example are: $\theta^{S_1}(\mathbf{i}) = (0, i, 0)^t$, $\theta^{S_2}(\mathbf{i}) = (0, i, 1, j, 0)$, $\theta^{S_3}(\mathbf{i}) = (1)$, $\theta^{S_4}(\mathbf{i}) = (2, k, 0)$, $\theta^{S_5}(\mathbf{i}) = (2, k, 1)$.

E.g., the Θ-matrices for S_2 and S_4 are:

$$\Theta^{S_2} = \begin{bmatrix} 0\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1 \\ 0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0 \end{bmatrix} \text{ with } \left| \begin{array}{l} \mathbf{i} = (i,j) \\ \mathbf{i}_{\text{lp}} = () \\ \mathbf{i}_{\text{gp}} = (N,M) \end{array} \right. \qquad \Theta^{S_4} = \begin{bmatrix} 0\,0\,0\,2 \\ 1\,0\,0\,0 \\ 0\,0\,0\,0 \end{bmatrix} \text{ with } \left| \begin{array}{l} \mathbf{i} = (k) \\ \mathbf{i}_{\text{lp}} = () \\ \mathbf{i}_{\text{gp}} = (N,M) \end{array} \right.$$

3.2 Invariants

Our representation makes a clear separation between the *semantically meaningful transformations* expressible on the polyhedral representation from the *semantically safe transformations* satisfying the statically checkable properties. The goal is of course to *widen the range of meaningful transformations* without relying on the accuracy of a static analyzer. Although classical transformations are hampered from the lack of information about loops bounds, they may be feasible in a polyhedral representation separating domains from affine schedules and authorizing per-statement operations. To reach this goal and to achieve a high degree of transformation compositionality, the representation enforces a few *invariants* on the domains and schedules.

There is only one domain invariant. To avoid integer overflows, the coefficients in a row of Λ^S must be relatively prime:

$$\forall 1 \le i \le d^S, \gcd(\Lambda_{i,1}, \ldots, \Lambda_{i,d_{\text{gp}}+1}) = 1. \tag{1}$$

This restriction has no effect on the expressible domains.

The first schedule invariant requires the schedule matrix to fit into a decomposition amenable to composition and code generation. It separates the square *iteration reordering matrix* $\mathrm{A}^S \in \mathcal{M}_{d^S,d^S}(\mathbb{Z})$ operating on iteration vectors, from the *parameterized matrix* $\Gamma^S \in \mathcal{M}_{d^S,d_{\text{gp}}+1}(\mathbb{Z})$ and from the *statement-scattering vector* $\beta^S \in \mathbb{N}^{d^S+1}$:

$$\Theta^S = \begin{bmatrix} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ \mathrm{A}^S_{1,1} & \cdots & \mathrm{A}^S_{1,d^S} & \Gamma^S_{1,1} & \cdots & \Gamma^S_{1,d_{\text{gp}}} & \Gamma^S_{1,d_{\text{gp}}+1} \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ \mathrm{A}^S_{2,1} & \cdots & \mathrm{A}^S_{2,d^S} & \Gamma^S_{2,1} & \cdots & \Gamma^S_{2,d_{\text{gp}}} & \Gamma^S_{2,d_{\text{gp}}+1} \\ \vdots & \ddots & \vdots & 0 & \ddots & 0 & \vdots \\ \mathrm{A}^S_{d^S,1} & \cdots & \mathrm{A}^S_{d^S,d^S} & \Gamma^S_{d^S,1} & \cdots & \Gamma^S_{d^S,d_{\text{gp}}} & \Gamma^S_{d^S,d_{\text{gp}}+1} \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta^S_{d^S} \end{bmatrix}. \tag{2}$$

Statement scattering may not depend on loop counters or parameters, hence the zeroes in "even dimensions". Notice β subscripts range from 0 to d^S.

Back to the running example, matrix Θ^{S_2} splits into

$$\mathrm{A}^{S_2} = \begin{bmatrix} 1\,0 \\ 0\,1 \end{bmatrix}, \Gamma^{S_2} = \begin{bmatrix} 0\,0\,0 \\ 0\,0\,0 \end{bmatrix}, \beta^{S_2} = (0,1,0)^t$$

The second schedule invariant is the *sequentiality* one: two distinct statement iterations may not have the same time-stamp:

$$S \neq S' \vee \mathbf{i} \neq \mathbf{i}' \Rightarrow \theta^S(\mathbf{i}) \neq \theta^{S'}(\mathbf{i}'). \tag{3}$$

Whether the iterations belong to the domain of S and S' does not matter in (3): we wish to be able to transform iteration domains without bothering with the sequentiality of the schedule. Because this invariant is hard to enforce directly, we introduce two additional invariants with no impact on schedule expressivity and stronger than (3):

$$|\det(\mathrm{A}^S)| = 1, \text{ i.e., } \mathrm{A}^S \text{ is unimodular, and } S \neq S' \Rightarrow \beta^S \neq \beta^{S'}. \tag{4}$$

Finally, we add a *density* invariant to avoid integer overflow and ease schedule comparison. The "odd dimensions" of the image of Θ^S form a d^S-dimensional sub-space of the multidimensional time, since A^S is unimodular, but an additional requirement is needed to enforce that "even dimensions" satisfy some form of dense encoding:

$$\beta^S_k > 0 \Rightarrow \exists S' \in \mathcal{S}, \mathrm{pfx}(\beta^S, k) = \mathrm{pfx}(\beta^{S'}, k) \wedge \beta^{S'}_k = \beta^S_k - 1, \tag{5}$$

i.e., for a given prefix, the next dimension of the statement-scattering vectors span an interval of non-negative integers.

3.3 Constructors

We define some elementary functions on SCoPs, called *constructors*. Many matrix operations consist in adding or removing a row or column. 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, whereas $\mathsf{RemRow}(\mathrm{M}, i)$ does the opposite transformation. Analogous constructors exist for columns, $\mathsf{AddCol}(\mathrm{M}, j, v)$ inserts a new column at position j in M and fills it with vector v, whereas $\mathsf{RemCol}(\mathrm{M}, j)$ undoes the insertion. AddRow and RemRow are extended to operate on vectors.

Displacement of a statement S is also a common operation. It only impacts the statement-scattering vector $\beta^{S'}$ of some statements S' sharing some common property with S. Indeed, forward or backward movement of S at depth ℓ triggers the same movement on every subsequent statement S' at depth ℓ such that $\mathrm{pfx}(\beta^{S'}, \ell) = \mathrm{pfx}(\beta^S, \ell)$. Although rather intuitive, the following definition with prefixed blocks of statements is rather technical. Consider a SCoP $\mathcal{S}$, a *statement-scattering prefix P* defining the depth at which statements should be displaced, a *statement-scattering prefix Q* — prefixed by P — making the initial time-stamp of statements to be displaced, and a displacement distance o; o is the value to be added/subtracted to the component at depth $\dim(P)$ of any statement-scattering vector β^S prefixed by P and following Q. The displacement constructor $\mathsf{Move}(P, Q, o)$ leave all statements unchanged except those satisfying

$$\forall S \in \mathcal{S}, P \sqsubseteq \beta^S \wedge (Q \ll \beta^S \vee Q \sqsubseteq \beta^S) : \beta^S_{\dim(P)} \leftarrow \beta^S_{\dim(P)} + o. \tag{6}$$

Constructors make no assumption about representation invariants and may violate them.

3.4 Primitives

From the earlier constructors, we will now define transformation *primitives* that enforce the invariants and serve as building blocks for higher level, semantically sound transformations. Most primitives correspond to simple polyhedral operations, but their formal definition is rather technical and will be described more extensively in a further paper. Figure 5 lists the main primitives affecting the polyhedral representation of a statement.[5] U denotes a unimodular matrix; M implements the parameterized shift (or translation) of the affine schedule of a statement; ℓ denotes the depth of a statement insertion, iteration domain extension or restriction; and c is a vector implementing an additional domain constraint.

The last two primitives — fusion and split (or distribution) — show the benefit of designing loop transformations at the abstract semantical level of polyhedra. First of all, loop bounds are not an issue since the code generator will handle any overlapping of iteration domains. Next, these primitives do *not* directly operate on loops, but consider prefixes P of statement-scattering vectors. As a result, they may virtually be composed with *any possible* transformation. For the split primitive, vector (P,o) prefixes all statements concerned by the split; 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 split (with the appropriate value of b) leaves the SCoP unchanged.

This table is not complete: privatization, array contraction and copy propagation require operations on access functions.

3.5 Transformation Composition

We will illustrate the composition of primitives on a typical example: two-dimensional tiling. To define such a composed transformation, we first build the strip-mining and interchange transformations from the primitives, as shown in Figure 6.

INTERCHANGE(S,o) swaps the roles of $\mathbf{i}_o$ and $\mathbf{i}_{o+1}$ in the schedule of S; it is a per-statement extension of the classical interchange. STRIPMINE(S,o,k) — where k is a *known integer* — prepends a new iterator to virtually k-times unroll the schedule and iteration domain of S at depth o. Finally, TILE(S,o,k) tiles the loops at depth o and $o+1$ with $k \times k$ blocks.

This tiling transformation is a first step towards a higher-level *combined* transformation, integrating strip-mining and interchange with privatization, array copy propagation and hoisting for dependence removal. The only remaining parameters would be the statements and loops of interest and the tile size.

4 Code Generation

After polyhedral transformations, code generation is the last step to the final program. It is often ignored in spite of its impact on the target code quality. In particular, we

[5] Many of these primitives can be extended to blocks of statements sharing a common statement-scattering prefix (like the fusion and split primitives).

Syntax & Name	Prerequisites	Effect
LEFTU (S,U) *Unimodular*	$S \in \mathcal{S} \wedge U \in \mathcal{M}_{d^S,d^S}(\mathbb{Z})$ $\wedge \lvert\det(\mathrm{U})\rvert = 1$	$\mathrm{A}^S \leftarrow \mathrm{U}.\mathrm{A}^S$
RIGHTU (S,U) *Unimodular*	$S \in \mathcal{S} \wedge U \in \mathcal{M}_{d^S,d^S}(\mathbb{Z})$ $\wedge \lvert\det(\mathrm{U})\rvert = 1$	$\mathrm{A}^S \leftarrow \mathrm{A}^S.\mathrm{U}$
SHIFT (S,M) *Shift*	$S \in \mathcal{S} \wedge \mathrm{M} \in \mathcal{M}_{d^S,d_{\mathrm{gp}}+1}(\mathbb{Z})$	$\Gamma^S \leftarrow \Gamma^S + \mathrm{M}$
INSERT (S,ℓ) *Insertion*	$\ell \leq d^S \wedge \beta^S_{\ell+1} = \cdots = \beta^S_{d^S} = 0$ $\wedge(\exists S' \in \mathcal{S}, \mathrm{pfx}(\beta^S,\ell+1) \sqsubseteq \beta^{S'}$ $\vee (\mathrm{pfx}(\beta^S,\ell), \beta^S_\ell - 1) \sqsubseteq \beta^{S'})$	$P = \mathrm{pfx}(\beta^S,\ell)$ $\mathcal{S} \leftarrow \mathsf{Move}(P,(P,\beta^S_\ell),1) \cup S$
DELETE (S) *Deletion*	$S \in \mathcal{S}$	$P = \mathrm{pfx}(\beta^S, d^S)$ $\mathcal{S} \leftarrow \mathsf{Move}(P,(P,\beta^S_{d^S}),-1) \setminus S$
EXTEND (S,ℓ) *Extension*	$S \in \mathcal{S}$	$d^S \leftarrow d^S + 1; \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S,\ell,0);$ $\mathrm{A}^S \leftarrow \mathsf{AddRow}(\mathsf{AddCol}(\mathrm{A}^S,\ell,0),\ell,1_\ell);$ $\beta^S \leftarrow \mathsf{AddRow}(\beta^S,\ell,0); \Gamma^S \leftarrow \mathsf{AddRow}(\Gamma^S,\ell,0);$ $\forall(\mathtt{A},\mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F},\ell,0)$
RESTRICT (S,ℓ) *Restriction*	$S \in \mathcal{S}$	$d^S \leftarrow d^S - 1; \Lambda^S \leftarrow \mathsf{RemCol}(\Lambda^S,\ell);$ $\mathrm{A}^S \leftarrow \mathsf{RemRow}(\mathsf{RemCol}(\mathrm{A}^S,\ell),\ell);$ $\beta^S \leftarrow \mathsf{RemRow}(\beta^S,\ell); \Gamma^S \leftarrow \mathsf{RemRow}(\Gamma^S,\ell);$ $\forall(\mathtt{A},\mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{RemRow}(\mathrm{F},\ell)$
CUTDOMAIN (S,c) *Cut Domain*	$S \in \mathcal{S}$ $\wedge \dim(c) = d^S + d^S_{\mathrm{lp}} + d_{\mathrm{gp}} + 1$	$\Lambda^S \leftarrow \mathsf{AddRow}(\Lambda^S, 0,$ $c/\gcd(c_1,\ldots,c_{d^S+d^S_{\mathrm{lp}}+d_{\mathrm{gp}}+1}))$
ADDLP (S) *Add Local Parameter*	$S \in \mathcal{S}$	$d^S_{\mathrm{lp}} \leftarrow d^S_{\mathrm{lp}} + 1;$ $\Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S, d^S + 1, 0);$ $\forall(\mathtt{A},\mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{AddCol}(\mathrm{F}, d^S + 1, 0)$
FUSE (P,o) *Fusion*		$b = \max\{\beta^S_{\dim(P)+1} \mid (P,o) \sqsubseteq \beta^S\} + 1;$ $\mathsf{Move}((P,o+1),(P,o+1),b);$ $\mathsf{Move}(P,(P,o+1),-1)$
SPLIT (P,o,b) *Split*		$\mathsf{Move}(P,(P,o,b),1);$ $\mathsf{Move}((P,o+1),(P,o+1),-b)$

Fig. 5. Main transformation primitives

Syntax & Name	Prerequisites	Effect	Comments
INTERCHANGE (S,o) *Loop Interchange*	$S \in \mathcal{S}$ $\wedge\, o < d^S$	$U = I_{d^S} - 1_{o,o} - 1_{o+1,o+1} + 1_{o,o+1} + 1_{o+1,o}$ $S \leftarrow$ RIGHTU (S,U)	swap rows o and $o+1$
STRIPMINE (S,o,k) *Strip Mining*	$S \in \mathcal{S}$ $\wedge\, o \leq d^S$ $\wedge\, k > 0$	$S \leftarrow$ EXTEND (S,o); $S \leftarrow$ ADDLP (S); $p = d^S + 1;$ $u = d^S + d^S_{\mathrm{lp}} + d_{\mathrm{gp}} + 1;$ $S \leftarrow$ CUTDOMAIN $(S, 1_{o+1} - 1_o)$; $S \leftarrow$ CUTDOMAIN $(S, 1_o - 1_{o+1} + (k-1)1_u)$; $S \leftarrow$ CUTDOMAIN $(S, 1_o - 1_p)$; $S \leftarrow$ CUTDOMAIN $(S, 1_p - 1_o)$;	 local param. column constant column $(\mathbf{i}_o \leq \mathbf{i}_{o+1})$ $(\mathbf{i}_{o+1} \leq \mathbf{i}_o + k - 1)$ $(k \times p \leq ii)$ $(ii \leq k \times p)$
TILE (S,o,k) *Tiling*	$S \in \mathcal{S}$ $\wedge\, o < d^S$ $\wedge\, k > 0$	$S \leftarrow$ STRIPMINE (S,o,k); $S \leftarrow$ STRIPMINE $(S,o+2,k)$; $S \leftarrow$ INTERCHANGE $(S,o+1)$;	

Fig. 6. Composition of transformation primitives

must ensure that a bad control management does not spoil performance, for instance by producing redundant guards or complex loop bounds.

Ancourt and Irigoin [1] proposed the first solution, based on the Fourier-Motzkin pair-wise elimination. The scope of their method was limited to a single polyhedron with unimodular transformation (scheduling) matrices. The basic idea was to apply the transformation function as a change of base of the loop indices, then for each new dimension, to project the polyhedron on the axis and thus find the corresponding loop bounds. The main drawback of this method was the large amount of redundant control. Most further works on code generation tried to extend this first technique, in order to deal with non-unit strides [15, 21] or with a non-invertible transformation matrix [9]. A few alternatives to Fourier-Motzkin were discussed, but without addressing the challenging problem of scanning more than one polyhedron at once.

This problem was first solved and implemented in Omega by generating a naive perfectly nested code and then by (partially) eliminating redundant guards [14]. Another way was to generate the code for each polyhedron separately, and then to merge them [9, 5]; it generates a lot of redundant control, even if there were no redundancies in the separated code. Quilleré et al. proposed to recursively separate union of polyhedra into subsets of disjoint polyhedra and generating the corresponding nests from the outermost to the innermost levels [19]. This approach provides at present the best solutions since it totally eliminates redundant control. However, it suffers from some limitations, e.g. high complexity, code generation with unit strides only, and a rigid partial order on the polyhedra. Improvements are presented in the next section.

This section presents the code generation problem, its resolution with a modern polyhedral-scanning technique, and its implementation.

4.1 The Code Generation Problem

In the polyhedral model, code generation amounts to a *polyhedron scanning problem*: finding a set of nested loops visiting each integral point, following a given scanning order. The generated code quality can be assessed by using two valuations: the most important is the amount of duplicated control in the final code; second, the code size, since a large code may pollute the instruction cache. We choose the recent Quilleré et al. method [19] with some additional improvements, which guarantee a code generation without any duplicated control. The outline of the modified algorithm is presented in Section 4.2 and some useful optimization are discussed in Section 4.3.

4.2 Outline of the Code Generation Algorithm

Our code generation process is divided in two main steps. First, we take the scheduling functions into account by modifying each polyhedron's lexicographic order. Next, we use an improved Quilleré et al. algorithm to perform the actual code generation.

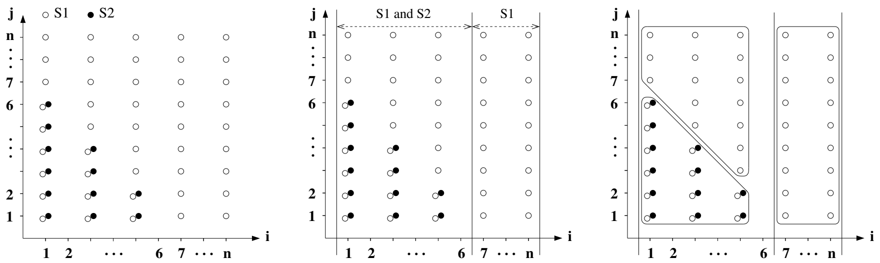
When no schedule is specified, the scanning order is the plain lexicographic order. Applying a new scanning order to a polyhedron amounts to adding new dimensions in leading positions. Thus, from each polyhedron $\mathcal{D}^S$ and scheduling function θ^S, we build another polyhedron $\mathcal{T}^S$ with the desired lexicographic order: $(\mathbf{t},\mathbf{i}) \in \mathcal{T}^S$ if and only if $\mathbf{t} = \theta^S(\mathbf{i})$. The algorithm is a recursive generation of the scanning code, maintaining a list of polyhedra from the outermost to the innermost loops:

1. intersect each polyhedron of the list with the context of the current loop (to restrict the scanning code to this loop);
2. project the resulting polyhedra onto the outermost dimensions, then separate the projections into disjoint polyhedra;

3. sort the resulting polyhedra such that a polyhedron is before another one if its scanning code has to precede the other to respect the lexicographic order;
4. merge successive polyhedra having at least another loop level to generate a new list and recursively generate the loops that scan this list;
5. compute the strides that the current dimension imposes to the outer dimensions.

This algorithm is slightly different from the one presented by Quilleré et al. in [19]; our two main contributions are the support for non-unit strides (Step 5) and the exploitation of degrees of freedom (i.e., when some operations do not have a schedule) to produce a more effective code (Step 4).

Let us describe this algorithm with a non-trivial example: the two polyhedral domains presented in Figure 7(a). Both statements have iteration vector (i, j), local parameter vector (k) and global parameter vector (n). We first compute intersections with the context, supposed to be $n \geq 6$. We project the polyhedra onto the first dimension, i, then separate them into disjoint polyhedra. Thus we compute the domains associated with $\mathcal{T}^{S_1}$ alone, both $\mathcal{T}^{S_1}$ and $\mathcal{T}^{S_2}$, and $\mathcal{T}^{S_2}$ alone (as shown in Figure 7(b), this last domain is empty). We notice there is a local parameter implying a non-unit stride; we can determine this stride and update the lower bound. We finally generate the scanning code for this first dimension. We now recurse on the next dimension, repeating the process for each polyhedron list (in this example, there are now two lists: one inside each generated outer loop). We intersect each polyhedra with the new context, now the outer loop iteration domains; then we project the resulting polyhedra on the outer dimensions, and finally we separate these projections into disjoint polyhedra. This last processing is trivial for the second list but yields two domains for the first list, as shown in Figure 7(c). Eventually, we generate the code associated with the new dimension.



$$\mathcal{T}_1^{S_1}(n) : \begin{cases} 1 \leq i \leq n \\ i = 2k+1 \\ 1 \leq j \leq n \end{cases}$$

$$\mathcal{T}_1^{S_2}(n) : \begin{cases} 1 \leq i \leq 6 \\ i = 2k+1 \\ 1 \leq j \leq 7-i \end{cases}$$

(a) Initial domains to scan

do i = 1, 6, 2
$\mathcal{T}_{1,1}^{S_1}(n) : \{1 \leq j \leq n\}$
$\mathcal{T}_{1,1}^{S_2}(n) : \{1 \leq j \leq 7-i\}$
do i = 7, n, 2
$\mathcal{T}_{1,2}^{S_1}(n) : \{1 \leq j \leq n\}$

(b) Projection and separation on the first dimension

code generation example

```
do i = 1, 6, 2
  do j = 1, 7-i
    S1; S2
  do j = 8-i, n
    S1
do i = 7, n, 2
  do j = 1, n
    S1
```

(c) Recursion on next dimensions

Fig. 7. Step by step code generation example

4.3 Complexity Issues

The main computing kernel in the code generation process is the separation into disjoint polyhedra, with a worst-case $O(3^n)$ complexity in polyhedral operations (exponential themselves). In addition, the memory usage is very high since we have to allocate memory for each separated domain. For both issues, we propose a partial solution. First of all, we use pattern matching to reduce the number of polyhedral computations: at a given depth, the domains are often the same (this is a property of the input codes), or disjoint (this is a property of the statement-scattering vectors of the scheduling matrices). Second, to avoid memory problems, we detect high memory consumption and switch for a more naive algorithm when necessary, leading to a less efficient code but using far less memory.

Our implementation of this algorithm is called CLooG (Chunky Loop Generator) and was originally designed for a locality-improvement algorithm and software (Chunky) [3]. CLooG could regenerate code for *all* 12 benchmarks in Figure 2. Experiments were conducted on a 512 *MB* 1 *GHz* Pentium III machine; generation times range from 1 to 127 seconds (34 seconds on average). It produced optimal control for all but three SCoPs in lucas, apsi and adm; the first SCoP has more than 1700 statements and could be optimally generated on a 1 *GB* Itanium machine in 22 minutes; the two other SCoPs have less than 50 statements, but 16 parameters; since the current version of does not analyse the linear relations between variables, the variability of parameter interactions leads to an exponential growth of the generated code. Complexity improvements and studies of the generated code quality are under investigation.

5 WRaP-IT: an Open64 Plug-In for Polyhedral Transformations

Our main goal is to streamline the extraction of static control parts and the code generation, to ease the integration of polyhedral techniques into optimizing and parallelizing compilers. This interface tool is built on Open64/ORC. It converts the WHIRL — the compiler's hierarchical intermediate representation — to an augmented polyhedral representation, maintaining a correspondence between matrices in SCoP descriptions with the symbol table and syntax tree. This representation is called the WRaP: WHIRL Represented as Polyhedra. It is the basis for any polyhedral analysis or transformation. Then, the second part of the tool is a modified version of CLooG, to regenerate a WHIRL syntax tree from the WRaP. The whole Interface Tool is called WRaP-IT; it may be used in a normal compilation or source-to-source framework, see [2] for details.

Although WRaP-IT is still a prototype, it proved to be very robust; the whole source-to-polyhedra-to-source transformation was successfully applied to all 12 benchmarks in Figure 2. See `http://www-rocq.inria.fr/a3/wrap-it` for further information.

6 Conclusion

We described a framework to streamline the design of polyhedral transformations, based on a unified polyhedral representation and a set of transformation primitives. It decouples transformations from static analyses. It is intended as a formal tool for semi-automatic optimization, where program transformations — with the associated static analyses for semantic-preservation — are separated from the optimization or parallelization algorithm which drives the transformations and select their parameters.

We also described WRaP-IT, a robust tool to convert back and forth between Fortran or C and the polyhedral representation. This tool is implemented in Open64/ORC. The complexity of the code generation phase, when converting back to source code, has long been a deterrent for using polyhedral representations in optimizing or parallelizing compilers. However, our code generator (CLooG) can handle loops with more than 1700 statements. Moreover, the whole source-to-polyhedra-to-source transformation was successfully applied to the 12 benchmarks. This is a strong point in favor of polyhedral techniques, even in the context of real codes.

Current and future work include the design and implementation of a polyhedral transformation library, an iterative compilation scheme with a machine-learning algorithm and/or an empirical optimization methodology, and the optimization of the code generator to keep producing optimal code on larger codes.

References

1. C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.
2. C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. Research report 4902, INRIA Rocquencourt, France, July 2003.
3. C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 Intl. Conference on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, Poland, april 2003.
4. 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.
5. P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3):421–444, 1998.
6. 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.
7. R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on Parallel and Distributed Systems*, 9(1):5–23, Jan. 1998.
8. P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. Journal of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, One Dimensional Time, 21(5):315–348.
9. M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *PACT'98 Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 106–111, 1998.
10. 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.
11. M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.
12. 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.
13. W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.
14. 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.
15. W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, April 1994.
16. 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.
17. M. O'Boyle, P. Knijnenburg, and G. Fursin. Feedback assisted iterative compiplation. In *Parallel Architectures and Compilation Techniques (PACT'01)*. IEEE Computer Society Press, Oct. 2001.
18. Open research compiler. `http://ipf-orc.sourceforge.net`.
19. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, october 2000.
20. 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.
21. J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.
22. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, San Diego, California, June 2003.