Code Generation in the Polyhedral Model Is Easier Than You Think

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

Abstract

Many advances in automatic parallelization and optimization have been achieved through the polyhedral model. It has been extensively shown that this computational model provides convenient abstractions to reason about and apply program transformations. Nevertheless, the complexity of code generation has long been a deterrent for using polyhedral representation in optimizing compilers. First, code generators have a hard time coping with generated code size and control overhead that may spoil theoretical benefits achieved by the transformations. Second, this step is usually time consuming, hampering the integration of the polyhedral framework in production compilers or feedback-directed, iterative optimization schemes. Moreover, current code generation algorithms only cover a restrictive set of possible transformation functions. This paper discusses a general transformation framework able to deal with non-unimodular, non-invertible, non-integral or even non-uniform functions. It presents several improvements to a state-of-the-art code generation algorithm. Two directions are explored: generated code size and code generator efficiency. Experimental evidence proves the ability of the improved method to handle real-life problems.

1. Introduction

Usual compiler intermediate representations like abstract syntax trees are not appropriate for complex program restructuring. Simple optimizations e.g. constant folding or scalar replacement may be achieved without hard modifications of such stiff data structures. But more complex transformations such as loop inversion, skewing, tiling etc. modify the execution order and this is far away from the syntax. A model based on a linear-algebraic representation of programs and transformations emerged in the Eighties to address this issue : the *polyhedral* (or *polytope*) *model*. This model became very popular because of its rich mathematical theory and its intuitive geometric interpretation. Moreover it adresses a class of codes with very regular control that includes a large range of real-life program parts [3].

The polyhedral framework is basically a plugin to the conventional compilation process. It starts from the abstract syntax tree by translating the program parts that fit the model into the linear-algebraic representation. The next step is to select a new execution order by using a reordering function (a schedule, or a placement, or a chunking function). Finding suitable execution orders has been the subject of most of the research on the polyhedral model [4, 5, 9, 12, 13, 20, 22, 24, 27]. Lastly the *code generation* step returns back to an abstract syntax tree or to a new source code implementing the execution order implied by the reordering function.

Up to now, the polyhedral model failed to integrate production compilers. Main reasons touch on the code generation step. Firstly, most algorithms require severe limitations on the reordering functions (e.g. to be unimodular or invertible) which reduce the opportunities of the optimization techniques to find efficient solutions. Next, simple-minded schemes for loop building may generate large and/or inefficient codes which can offset the optimization they are enabling. Finaly, the complexity of the problem is challenging for real-life programs and hampers the integration of the framework in iterative optimization schemes. In this paper, we will show how it is possible to handle very general transformations in the polyhedral model and that starting from one of the best algorithms known so far [21], how we can improve it for producing in a reasonnable amount of time an efficient target code with a limited size growing.

The paper is organized as follows. Section 2 introduces the polyhedral model formaly. Section 3 presents a general program transformation framework within this model. Section 4 describes the code generation algorithm and proposes new ways to achieve quickly an efficient, small target code. In section 5, experimental results obtained through the algorithm implementation are shown. Finally, section 6 discusses related work and section 7 summarizes the main contributions of this paper then discusses future works.

2. Background and Notations

The polyhedral model is a representation of both sequential and parallel programs. It corresponds to a subset of imperative languages like C or FORTRAN known as *static control programs* [11]. This class includes a large range of programs which are discussed in depth by Xue [28]. Their properties can be roughly summarized in this way: (1) control statements are **do** loops with affine bounds and **if** conditionals with affine conditions (in fact control can be more complex, see [28]); (2) affine bounds and conditions depend only on outer loop counters and constant parameters. A maximal set of consecutive statements with static control in any program is called a static control part (SCoP). The kernel in Figure 1 is an example of strict acceptance to the static control restrictions and will be used for illustrating further concepts. The loops in such an imperative

```
      do i=1, n
S1    | x = a(i,i)
      | do j=1, i-1
S2    | | x = x - a(i,j)**2
S3    | p(i) = 1.0/sqrt(x)
      | do j=i+1, n
S4    | | x = a(i,j)
      | | do k=1, i-1
S5    | | | x = x - a(j,k)*a(i,k)
S6    | | a(j,i) = x*p(i)
```

Figure 1. A Cholesky factorization kernel

language can be represented using n-entry column vectors called *iteration vectors*: $\vec{x} = (i_1, i_2, \ldots, i_n)^T$, where i_k is the k^{th} loop index and n is the innermost loop. Considering the static control class, the program execution can be fully described by using two specifications for each statement:

- The *iteration domain* $\mathcal{D}$, i.e. the set of values of the iteration vector for which the statement has to be executed. When a statement is surrounded with static control, its iteration domain can always be specified by a set of linear inequalities defining a polyhedron [16]. The term *polyhedron* will be used in a broad sense to denote a *convex set of points in a lattice* (also called $\mathbb{Z}$-polyhedron or lattice-polyhedron), i.e. a set of points in a $\mathbb{Z}$ vector space bounded by affine inequalities [23]:

$$\mathcal{D} = \{\vec{x} \mid \vec{x} \in \mathbb{Z}^n, A\vec{x} \geq \vec{c}\},$$

  where $\vec{x}$ is the iteration vector, A is a constant matrix and $\vec{c}$ is a constant vector, possibly parametric. Figure 3(a) illustrates the correspondence between static control and polyhedral domains for the statement $S2$ of the program in Figure 1.

- A *scattering function* $\theta(\vec{x})$, an affine function specifying for each integral point in the iteration domain a new coordinate for the corresponding statement instance:

$$\theta(\vec{x}) = T\vec{x} + \vec{t},$$

  where T is a constant matrix, and $\vec{t}$ is a constant vector, possibly parametric. Depending on the context, the scattering function may have several interpretations: to distribute the iterations in space, i.e. across different processors, to order them in time, or both (by composition), etc. In the case of *space-mapping*, the number returned by the function for a given statement instance is the number of the processor where it has to be executed. In an n-dimensional *time-schedule*, the statement instance with the logical date $(a_1...a_n)$ is executed before those with associated date $(b_1...b_n)$ iff $\exists i, 1 \leq i < n, (a_1...a_i) = (b_1...b_i) \wedge a_{i+1} < b_{i+1}$, i.e. they follow the *lexicographic order*. For instance we can easily capture the sequential execution order of any static control program with scheduling functions by using the abstract syntax tree of this program [12]: we can read directly such functions for the program in Figure 1 on the AST shown in Figure 2, e.g. $\theta_{S1}(\vec{x}_{S1}) = (0, i, 0)^T$, $\theta_{S2}(\vec{x}_{S2}) = (0, i, 1, j, 0)^T$, $\theta_{S3}(\vec{x}_{S3}) = (0, i, 2)^T$ etc.

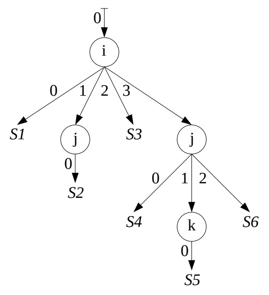
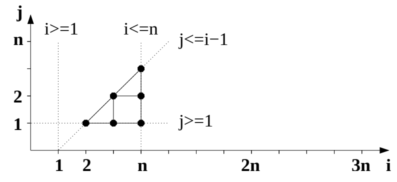


Figure 2. AST of the program in Figure 1

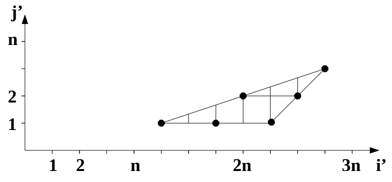
3. Program Transformations

Program transformations in the polyhedral model can be specified by well chosen scattering functions. They modify the source polyhedra into target polyhedra containing the same points but in a new coordinate system, thus with a new lexicographic order. Implementing these transformations is the central part of the polyhedral framework. The current polyhedral code generation algorithms lack flexibility by addressing only a subset of the possible functions. How to



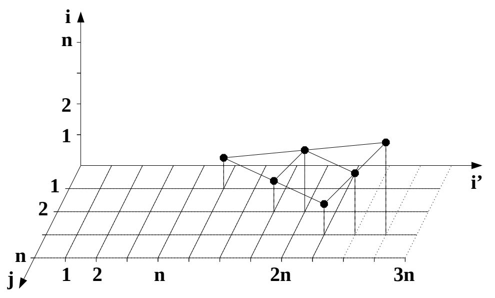
$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ 1 \end{pmatrix}$$

(a) original polyhedron $A\vec{x} \geq \vec{c}$



$$\begin{bmatrix} 1/2 & -1/2 \\ -1/2 & 1/2 \\ 0 & 1 \\ 1/2 & -3/2 \end{bmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ 1 \end{pmatrix}$$

(b) usual transformation $(AT^{-1})\vec{y} \geq \vec{c}$



$$\left[\begin{array}{c|cc} 1 & -2 & -1 \\ \hline 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{array}\right] \left(\begin{array}{c} i' \\ \hline \\ i \\ j \\ \\ \end{array}\right) \begin{array}{c} = \\ \hline \\ \geq \\ \\ \\ \end{array} \left(\begin{array}{r} 0 \\ \hline 1 \\ -n \\ 1 \\ 1 \end{array}\right)$$

(c) our transformation $\mathcal{T}$

Figure 3. Transformation policies for $\mathcal{D}_{S2}$ in Figure 1 with $\theta_{S2}(i,j) = 2i + j$

use general affine scattering functions to apply a new lexicographic order to the original polyhedra is explained in section 3.1. Section 3.2 and section 3.3 respectively deal with the special case of non-integral and non-uniform transformations and show how it is possible to handle them in this framework.

3.1. Affine Transformations

Previous work on code generation in the polyhedral model required severe limitations on the scattering functions, e.g. to be unimodular [1, 17] (the T matrix has to be square and has determinant ± 1) or at least to be invertible [20, 27, 22, 5]. The underlying reason was, considering an original polyhedron defined by $A\vec{x} \geq \vec{c}$ and the scattering function leading to the target index $\vec{y} = T\vec{x}$, the polyhedron in the new coordinate system is defined by $(AT^{-1})\vec{y} \geq \vec{c}$, a change of basis. Griebl et al. proposed the first relaxation of the invertibility constraint, by using a square invertible extension of the transformation matrix [14]. Unfortunately their method led practically to a very high control overhead.

In this paper we do not impose any constraint on the transformation functions because we do not try to perform a change of basis of the original polyhedron to the target index. Instead, we apply a new lexicographic order to the polyhedra by adding new dimensions in leading positions. Thus, from each polyhedron $\mathcal{D}$ and scattering function θ, it is possible to build another polyhedron $\mathcal{T}$ with the appropriate lexicographic order:

$$\mathcal{T} = \left\{ \left(\frac{\vec{y}}{\vec{x}} \right) \;\middle|\; \left[\begin{array}{c|c} Id & -T \\ \hline 0 & A \end{array} \right] \left(\frac{\vec{y}}{\vec{x}} \right) \frac{=}{\geq} \left(\frac{\vec{t}}{\vec{c}} \right) \right\},$$

where by definition, $(\vec{y}, \vec{x}) \in \mathcal{T}$ if and only if $\vec{y} = \theta(\vec{x})$. The points inside the new polyhedron are ordered lexicographically until the last dimension of $\vec{y}$. Then there is no particular order for the remaining dimensions.

By using such a transformation policy, the data of both original iteration domains and transformations are included in the new polyhedra. As an illustration, let us consider the polyhedron $\mathcal{D}_{S2}$ in Figure 3(a) and the scattering function $\theta_{S2}(i,j) = 2i+j$. The corresponding scattering matrix $T = [\,2\ \ 1\,]$ is not invertible, but it can be extended to $T = \left[\begin{smallmatrix} 2 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ as suggested by Griebl et al. [14]. The usual resulting polyhedron is shown in Figure 3(b). Our policy leads directly to the polyhedron in Figure 3(c), provided we choose the lexicographic order for the free dimensions. A projection onto i' and i would lead to the result in Figure 3(b). The additional dimensions carry the transformation data, i.e. in this case $j = i' - 2i$. This is helpful since during code generation we have to update the references to the iterators in the loop body, and *necessary* when the transformation is not invertible. Another property of this transformation policy is never to build rational target constraint systems. Most previous works were challenged by this problem, which occurs when the transformation function is non-unimodular. We can observe the phenomenon in Figure 3(b). The integer points without heavy dots have no images in the original polyhedron. The original coordinates can be determined from the target ones by $\overrightarrow{original} = T^{-1}\overrightarrow{target}$. Because T is non-unimodular, T^{-1} has rational elements. Thus some integer target points have a rational image in the original space; they are called *holes*. To avoid considering the holes, the strides (the steps between the integral points to consider) had to be found. Many works proposed to use the Hermite Normal Form [23] in different ways to solve the problem [20, 27, 9, 22]. In the opposite, we do not change the basis of the original polyhedra, but we only apply an appropriate lexicographic order. As a consequence, our target systems are always integral and there are no holes in the corresponding polyhedra. The stride informations are explicitly contained in the constraint systems in the form of equations.

The cost of this method is to add new dimensions to the polyhedra. This can be a relevant issue since first, it increases the complexity of the scanning step and second, it increases the constraint system size while high-level code generation typically requires a lot of memory. In practice processing of additional dimensions is often trivial with the method presented in section 4. Eventually, our prototype is more efficient and needs less memory than those based on other methods (see section 5).

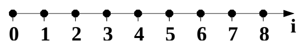
3.2. Rational Transformations

Some automatic allocators or schedulers ask for rational transformations [12]. Thus scattering functions can have a more general shape:

$$\theta(\vec{x}) = (T\vec{x} + \vec{t})/\vec{d},$$

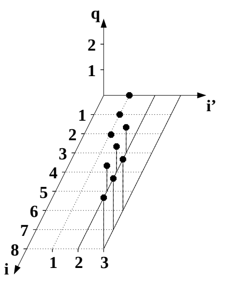
where $/$ means integer division and $\vec{d}$ is a constant vector such that each element divides the corresponding dimension of $\theta(\vec{x})$. In practice, divisors often correspond to resource constraints (e.g. the number of processors, of functional units etc.). Wetzel proposed the first solution to solve this problem, but only for one divisor value for the whole scattering function, and leading to a complex control [25].

Again, we propose to add dimensions to solve the problem. For each rational element in $(T\vec{x})/\vec{d}$, we introduce an auxiliary variable standing for the quotient of the division. For instance let us consider the original polyhedron in Figure 4(a) and the scheduling function $\theta(i) = i/3 + 1$. We introduce q and r such as $i = 3q + r$, with by definition $0 \leq r = i - 3q \leq 2$. Then we can deal with an equivalent integral transformation $\theta'(q) = q + 1$ with $0 \leq i - 3q \leq 2$. This amounts to *strip-mine* the dimension i, as shown in Figure 4(b). With several non-integer coefficients, we just need more auxiliary variables standing for the result of the divisions.



$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} (i) \geq \begin{pmatrix} 0 \\ -8 \end{pmatrix}$$

(a) original polyhedron $A\vec{x} \geq \vec{c}$



$$\left[\begin{array}{c|cc} 1 & -1 & 0 \\ \hline 0 & -3 & 1 \\ 0 & 3 & -1 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{array}\right] \left(\begin{array}{c} i' \\ \hline \\ q \\ i \\ \\ \end{array}\right) \begin{array}{c} = \\ \hline \\ \geq \\ \\ \\ \end{array} \left(\begin{array}{c} 1 \\ \hline 0 \\ -2 \\ 0 \\ -8 \end{array}\right)$$

(b) our transformation $\mathcal{T}$

Figure 4. Rational reordering $\theta(i) = i/3 + 1$

3.3. Non-Uniform Transformations

As the power of program analysis increased with time, program transformations became more and more complex in order to face new optimization opportunities. Starting from simple transformation for a single loop nest, they evolved to statement-wise functions and more recently to several transformations per statement, each of them applying to a subset of the iteration domain. Thus a scattering function for a statement with the iteration domain $\mathcal{D}$ may be of the following form:

$$\theta(\vec{x}) = \begin{cases} \text{if } \vec{x} \in \mathcal{D}_1 \text{ then } T_1\vec{x} + \vec{t_1} \\ \text{if } \vec{x} \in \mathcal{D}_2 \text{ then } T_2\vec{x} + \vec{t_2} \\ \ldots \\ \text{if } \vec{x} \in \mathcal{D}_n \text{ then } T_n\vec{x} + \vec{t_n} \end{cases}$$

where the $\mathcal{D}_i$, $1 \leq i \leq n$ are a partition of $\mathcal{D}$. It is quite simple to handle such transformations, at least when the code generator deals efficiently with more than one polyhedron, by explicitly splitting the considered polyhedra into partitions. When the iteration domain is split using affine conditions, as in *index set splitting* [13], building the partition is trivial, but more general partitions with non-affine criteria are possible as long as we can express each subset as a polyhedron. For instance, Slama et al. found programs where the best parallelization requires non-uniform transformations, e.g. $\theta(i) =$ if $(i \ mod \ d = n)$ then ... else ... where d is a scalar value and n a constant possibly parametric. They propose a code generation scheme dedicated to this problem [24]. It is possible to handle this in our framework by adding new dimensions. For instance the iteration domain corresponding to the *then* part of $\theta(i)$ would be the original one with the additional constraint $i = jd + n$, while the additional constraints for the *else* part could be $i \leq jd + n - 1$ and $i \geq jd + n + 1 - d$. Then we can apply the transformations to the resulting polyhedra as shown in section 3.1.

4. Scanning Polyhedra

We showed in previous sections that any static control programs can be specified using a set of iteration domains and scattering functions that can be merged to create new polyhedra with the appropriate lexicographic order. Generating code in the polyhedral model amounts to finding a set of nested loops visiting each integral point of each polyhedra, once and only once, following this order. This is a critical step in the framework since the final program effectiveness highly depends on the target code quality. In particular, we must ensure that a bad control management does not spoil performance, for instance by producing redundant conditions, complex loop bounds or under-used iterations. On the other hand, we have to avoid code explosion for instance because a large code may pollute the instruction cache.

At present, the Quilleré et al. method give the best results when we have to generate a scanning code for several polyhedra [21, 2]. This technique is guaranteed to avoid redundant control while scanning the scattering dimensions. However, it suffers from some limitations, e.g. high complexity and needless code explosion. In the following, we propose some solutions to these drawbacks. We present the general algorithm with some adaptations to our purpose in section 4.1. We address the problem of reducing the code size without consequence on code efficiency in section 4.2. Finally in section 4.3 we discuss complexity issues.

4.1. Extended Quilleré et al. Algorithm

Quilleré et al. proposed recently the first code generation algorithm building the target code without redundant control directly instead of starting from a naive code and trying to improve it [21]. As a consequence, this method never fail to remove a guard and the processing is easier. Eventually it generates a better code more efficiently. The algorithm rely on polyhedral operations that can be implemented by e.g. PolyLib[1] [26]. The basic mechanism is, starting from the list of polyhedra to scan, to recursively generate each level of the abstract syntax tree of the scanning code (AST). The nodes of the AST are labelled with a polyhedron $\mathcal{T}$ and have a list of children (notation $\mathcal{T} \rightarrow (...)$). The leaves are labelled with a polyhedron and a statement (notation $\mathcal{T}_S$). Each recursion builds an AST node list as described by the algorithm in Figure 5. It starts with the following input: (1) the list of transformed polyhedra to be scanned $(\mathcal{T}_{S_1}, ..., \mathcal{T}_{S_n})$; (2) the context, i.e. the set of constraints on the global parameters; (3) the first dimension $d = 1$. Generating the code from the AST is a trivial step: the constraint system labelling each node can be directly translated as loop bounds and as surrounding conditional, respectively if the constraints concern the dimension corresponding to the node level or not.

This algorithm is somewhat different from the one presented by Quilleré et al. in [21] and its improved version in [2]; our two main contributions are the following: reducing the code size without degrading code performance (step 7) and reduction of the code generation processing time by using pattern matching (step 3).

We propose to illustrate this algorithm (without step 7) through the example in Figure 6. We have to generate the scanning code for the three polyhedra in Figure 6(a). For the sake of simplicity, we will show directly the translations of the node constraint systems into source code. We first compute the intersections with the context (i.e., at this point, the constraints on the parameters, supposed to be $n \geq 2$ and $m \geq n$). We project the polyhedra onto the first dimension, i, then we separate them into disjoint polyhedra. As shown in Figure 6(b) this results in two disjoint polyhedra. We can now generate the scanning code for this first dimension. Then we 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 polyhedron with the new context, i.e. the outer loop iteration domains; then we project the resulting polyhedra onto the outer dimensions. Finally we separate these projections into disjoint polyhedra. This last process is trivial for the second list but yields several domains for the first list, as shown in Figure 6(c). Then we generate the code associated with the new dimension, and since this is the last one, a scanning code is fully generated. Lastly, we remove dead code (for instance in the first loop nest in Figure 6(c), the iteration $i = n$ is useful only for a small part of the loop body) by applying a new projection step during the recursion backtrack. The final code is shown in Figure 6(d).

4.2. Reducing code size

The power of optimizing methods in the polyhedral model are of a particular interest for embedded system compiling. One of the main constraint for such applications is the object code size because of inherent hardware limitations. Generated code size may be under control for this purpose or simply to avoid instruction cache pollution. It is possible to manage it easily with iterative code generation methods [15]: they start from a naive (inefficient) and short code and eliminate the control overhead by selecting conditions to remove and performing code hoisting (splitting the code on the chosen condition and copying the original guarded code in the two branches). Thus, to stop code hoisting stops code growing. With recursive code generation methods as discussed in this paper, it is always possible to choose not to separate the polyhedra and to generate

1 PolyLib is available at http://icps.u-strasbg.fr/PolyLib

CodeGeneration: Build a polyhedra scanning code without redundant control AST.

Input: a polyhedron list $(\mathcal{T}_{S_1}, ..., \mathcal{T}_{S_n})$, a context C, the current dimension d.
Output: the abstract syntax tree of the code scanning the polyhedra inside the input list.

1. Intersect each polyhedron $\mathcal{T}_{S_i}$ in the list with the context C in order to restrict the domain (and subsequently the code that will be generated) under the context of the surrounding loop nest.
2. Compute for each resulting polyhedron $\mathcal{T}_{S_i}$ its projection $\mathcal{P}_i$ onto the outermost d dimensions and consider the new list of $\mathcal{P}_i \rightarrow \mathcal{T}_{S_i}$.
3. Separate the projections into a new list of disjoint polyhedra: given a list of m polyhedra, start with the first two polyhedra $\mathcal{P}_1 \rightarrow \mathcal{T}_{S_1}$ and $\mathcal{P}_2 \rightarrow \mathcal{T}_{S_2}$ by computing $(\mathcal{P}_1 - \mathcal{P}_2) \rightarrow \mathcal{T}_{S_1}$ (i.e. S_1 alone), $(\mathcal{P}_1 \cap \mathcal{P}_2) \rightarrow (\mathcal{T}_{S_1}, \mathcal{T}_{S_2})$ (i.e. S_1 and S_2) and $(\mathcal{P}_2 - \mathcal{P}_1) \rightarrow \mathcal{T}_{S_2}$ (i.e. S_2 alone), then for the three resulting polyhedra, make the same separation with $\mathcal{P}_3 \rightarrow \mathcal{T}_{S_3}$ and so on.
4. Build the lexicographic ordering graph where there is an edge from a polyhedron $\mathcal{P}_1 \rightarrow (\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$ to another polyhedron $\mathcal{P}_2 \rightarrow (\mathcal{T}_{S_v}, ..., \mathcal{T}_{S_w})$ if its scanning code has to precede the other to respect the lexicographic order, then sort the list according to a valid order.
5. For each polyhedron $\mathcal{P} \rightarrow (\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$ in the list:
   (a) Compute the stride that the inner dimensions impose to the current one, and find the lower bound by looking for stride constraints in the $(\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$ list.
   (b) While there is a polyhedron in $(\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$:
      i. Merge successive polyhedra with another dimension to scan in a new list.
      ii. Recurse for the new list with the new loop context $C \cap \mathcal{P}$ and the next dimension $d + 1$.
6. For each polyhedron $\mathcal{P} \rightarrow$ (inside) in the list, apply steps 2 to 4 of the algorithm to the *inside* list in order to remove dead code. Then consider the concatenation of the resulting lists as the new list.
7. Make all the possible unions of *host* polyhedra with *point* polyhedra to reduce code size.
8. Return the polyhedron list.

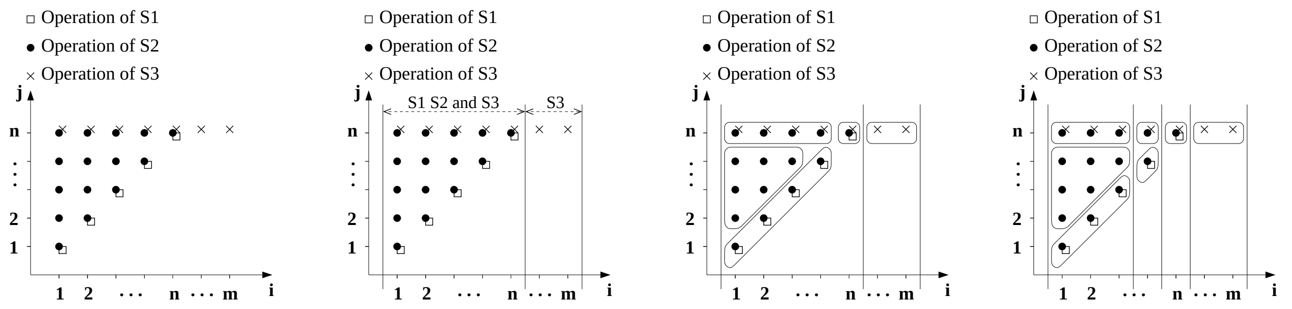
Figure 5. Extended Quilleré et al. Algorithm

a smaller code with conditions [21]. These techniques always operate at the price of a less efficient generated code. This section presents another way, with a quite small impact on control overhead and a possibly significant code size improvement. It is based on a simple observation: separating polyhedra often results in isolating some points, while this is not always necessary. Figure 6 shows a dramatic example of this phenomenon (hoisting-based code generators as the Omega CodeGen have to meet the same issue). Integrating these points inside *host* loops when possible will reduce the code size by adding new iterations. The problem was first pointed out by Bouchebaba in the particular case of 2-dimensional loop nest fusion [4]. He extracted the 14 situations where a vertex should not be fused with a loop for his purpose and apply the fusion in the other cases. In the following is presented a solution for general code generation based on the properties of the code construction algorithm in Figure 5.

To ensure that the separation step will not result in needless polyhedron peeling, it is necessary to compute this separation. In addition we have to achieve the recursion on every dimensions since the projection hide some of them during the separation process. Thus, we can remove isolated points at the end of each recursion (step 7). At a given depth of the recursion, the removing process is applied for each *loop node* in the list (i.e. such that the dimension corresponding to the current depth is not constant):

1. Define the point candidate to merge with the node: scan the node branch in depth first order and build the list of statements in the leaves. The statement candidate has to fit this statement list since it is guaranteed after dead code elimination that each statement in the leaves is executed at least once. Thus only a point with this structure may be merged with the node.
2. Check if such a point directly precedes or follows the node in the lexicographic ordering graph built in step 4 and 6 (details on this graph construction can be found in [21]). This graph is only based on the projecting dimensions, however if a point candidate directly follows the node in the ordering graph and cannot be merged, this means that an input polyhedron is not convex, a contradiction.
3. Merge the point candidates with the node if the previous test was a success by using a polyhedral *union*, and remove the points from the list of polyhedra.

We can apply this process to the example in Figure 6. The translation of the AST after dead code removing for the di-



$$\mathcal{T}_{S_1} : \begin{cases} 1 \le i \le n \\ j = i \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} 1 \le i \le n \\ i \le j \le n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \le i \le m \\ j = n \end{cases}$$

(a) Initial domains to scan

```
do i=1, n
```

$$\mathcal{T}_{S_1} : \begin{cases} 1 \le i \le n \\ j = i \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} 1 \le i \le n \\ i \le j \le n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \le i \le n \\ j = n \end{cases}$$

```
do i=n+1, m
```

$$\mathcal{T}_{S_3} : \begin{cases} n+1 \le i \le m \\ j = n \end{cases}$$

(b) Projection and separation onto the first dimension

```
do i=1, n
| if (i==n) then
| | S1(j=n)
| | S2(j=n)
| | S3(j=n)
| if (i<=n-1) then
| | S1(j=i)
| | S2(j=i)
| do j=i+1, n-1
| | S2
| if (i<=n-1) then
| | S2(j=n)
| | S3(j=n)
do i=n+1, m
| S3(j=n)
```

(c) Recursion on next dimension

```
do i=1, n-2
| S1(j=i)
| S2(j=i)
| do j=i+1, n-1
| | S2
| S2(j=n)
| S3(j=n)
S1(i=n-1,j=n-1)
S2(i=n-1,j=n-1)
S2(i=n-1,j=n)
S3(i=n-1,j=n)
S1(i=n,j=n)
S2(i=n,j=n)
S3(i=n,j=n)
do i=n+1, m
| S3(j=n)
```

(d) Backtrack with dead code removing

Figure 6. Step by step code generation example

mension j is equivalent to the code in Figure 6(c). The statement candidate for the j loop is S_2. We can merge both S_2 points before and after this loop. Then the dead code removing for dimension i would only isolate the point corresponding to $i = n$, the new candidate would be $S_1S_2S_3$. It can be merged and the final code is shown in Figure 7 with an object code size of 176B while the previous one in Figure 6(d) is 464B (each statement is a 2-dimensional array entry increment).

```
do i=1, n
| S1(j=i)
| do j=i, n
| | S2
| S3(j=n)
do i=n+1, m
| S3(j=n)
```

Figure 7. Compacted code of Figure 6(d)

4.3. Complexity Issues

The main computing kernel in the code generation process is the separation into disjoint polyhedra (step 3). Given a list of n polyhedra, the worst-case complexity is $\mathcal{O}(3^n)$ 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.

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, this happens for 17% of the operations in the benchmark set presented in section 5), or disjoint (this is a property of the scheduling matrices, this happens for 36% of the operations in the benchmark set of section 5). Thus we check quickly for these properties before any polyhedral operation by comparing directly the elements of the constraint systems (this allows to find 75% of the equalities), and by comparing the unknowns having fixed values (this allows to find 94% of the disjunctions). When one of these properties is proved, we can directly give the trivial solution to the operation. This method improves performance by a factor near to 2.

To avoid a memory allocation explosion, when we detect a high memory consumption, we continue the code generation process for the remaining recursions with a more naive algorithm, leading to a less efficient code but using far less memory. Instead of separating the projections into disjoint polyhedra (step 3 of the algorithm), we merge them when their intersections are not empty. Then we work with a set of unions, significantly smaller than a set of disjoint polyhedra. Other parts of the algorithm are left unmodified. The drawback of this method is the generation of costly conditionals ruling whether an integral point has to be scanned or not. This method can be compared to using the convex hull of the polyhedra [15, 25, 14, 5], but is more general since it can deal with complex bounds (typically maximum or minimum of parameterized affine constraints e.g. $max(m, n)$) that do not describe a convex polyhedron.

5. Experimental Results

We implemented this algorithm and integrated it into a complete polyhedral transformation infrastructure inside Open64/ORC [3]. Such a modern compiler provides many steps enabling the extraction of large static control parts (e.g. function inlining, loop normalization, `goto` elimination, induction variable substitution etc.). In this section is presented a study on the applicability of the presented framework to large, program representative SCoPs that have been extracted from SPECfp2000 and PerfectClub benchmarks. The chosen methodology was to perform the code regeneration of all these static control parts.

Figure 8 gives some informations on the code regeneration problem for a set of SPECfp 2000 and PerfectClub benchmarks. The first two columns gives the total number of SCoPs and iteration domains in the corresponding benchmark. These problems are considered to be hard: previously related experiences with Omega [15] or LooPo [14] showed how it was challenging to producing efficient code just for ten or so polyhedra without time or memory explosion. The two columns of the *code generation* section shows how many SCoPs have to be partially regenerated in a sub-optimal way because of a memory explosion and the total code generation time on a Intel Pentium III 1 GHz architecture with 512 MB RAM. The three challenging problems have a lot of free parameters (13 or 14) that leads to a high code versioning; the biggest one in lucas (more than 1700 domains) took 22 minutes and 1 GB RAM to be optimally generated on a Itanium 1 GHz machine. These results are very encouraging since the code generator proved its ability to regenerate real-life problems with hundreds of statements and a lot of free parameters. Both code generation time and memory requirement are acceptable in spite of a worst-case exponential algorithm complexity.

	SCoPs		Code Generation		Robustness	
	Total	Domains	Sub.	Time (s)	CodeGen	LoopGen
applu	25	757	0	28.16	39%	53%
apsi	109	2192	1	42.13	98%	98%
art	62	499	0	1.50	99%	100%
equake	40	639	0	6.80	73%	73%
lucas	11	2070	1	47.58	1%	1%
mgrid	12	369	0	4.53	54%	54%
swim	6	123	0	0.58	100%	100%
adm	109	2260	1	43.94	92%	92%
dyfesm	112	1497	0	14.81	84%	86%
mdg	33	530	0	4.52	82%	100%
mg3d	63	1442	0	18.56	85%	85%
qcd	74	819	0	28.23	79%	86%

Figure 8. Code generation of static control parts in high-performance applications

We compared the results achieved by our code generator, CLooG[2], with a previous implementation of the Quilleré et al. algorithm, LoopGen 0.4 [21] (the differences between CLooG and LoopGen are a direct consequence of the improvements discussed in this paper), and the most widely used code generator in the polyhedral model, i.e. Omega's CodeGen 1.2 [15]. Because of inherent limitations (mainly memory explosion), these generators are not able to deal with all the real-life code generation problems in the benchmark set, the section *robustness* in Figure 8 gives the percentages of the input problems they are able to deal with[3]. These results illustrate the existing need for scalability of code generation schemes. Hence, comparisons are done with the only common subset. The two valuations are the code generation time and the generated code size with respect to the original code size. The results are given in Figure 9. It shows that generating directly a code without redundant control is far more efficient than trying to improve a naive one. Our pattern matching strategy demonstrates its effectiveness, since we observe a significant speedup of 4.05 between CLooG and CodeGen and of 2.57 between CLooG and LoopGen. Generated code sizes by LoopGen are typically greater than CodeGen results by 38% on average because it removes more control overhead at the price of code size. The code size improvement methodology presented in this paper significantly reduces this increase to 6% on average while keeping up the generated code effectiveness.

In conclusion, our algorithm is much faster than CodeGen and noticeably faster than LoopGen. LoopGen generates larger code, while our code and the CodeGen code are

2 CLooG is available at `http://www.prism.uvsq.fr/∼cedb`

3 We only consider the code generation ability: for technical reasons, we did not check the correctness of Omega's CodeGen results.

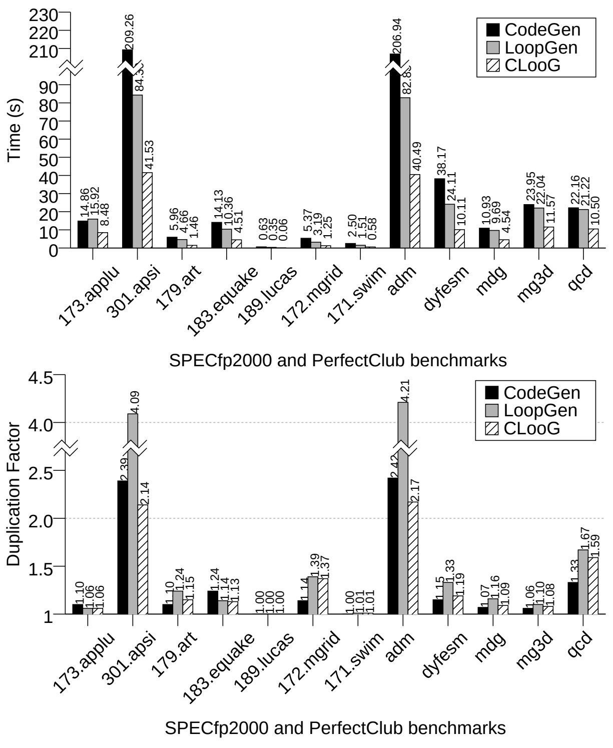


Figure 9. Code generation times and sizes

of about the same size. It remains to compare the run time overheads: our code has the same performance as the original code, and we believe this should be true also for LoopGen. For technical reasons, assessing the performances of CodeGen is difficult, and is left for future work.

6. Related Work

Ancourt and Irigoin [1] proposed the first solution to the polyhedron scanning problem. Their seminal work was based on the Fourier-Motzkin pair-wise elimination [23]. The scope of their method was very restrictive, since it could be applied to only one polyhedron, with unimodular transformation (scattering) matrices. The basic idea was to apply the transformation function as a change of basis of the loop index, then for each new dimension, to project the polyhedron onto 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 more general transformations. Li and Pingali [20], Xue [27], Darte [9] and Ramanujam [22] relaxed the unimodularity constraint to an invertibility constraint and then proposed to deal with non-unit strides (loop increments can be something different than one). They all use the Hermite Normal Form [23] to find the strides, and the classical Fourier-Motzkin elimination to compute the loop bounds. In addition, Li and Pingali proposed a completion algorithm to build a non-unimodular transformation function from a partial matrix, such as the transformation stay legal for dependences [20]. In the same spirit, Griebl et al. relaxed the invertibility constraint and proposed to deal with arbitrary matrix by using a square invertible extension of this matrix [14]. It is shown in this paper how to deal with general affine transformation functions without constraints on unimodularity, invertibility or even regularity.

Alternatively to the Fourier-Motzkin elimination method, Collard et al. [7] presented a loop bound calculation technique based on a parameterized version of the dual simplex algorithm [10]. Another method makes successive projections of the polyhedron on the axis as in [1] but use the Chernikova algorithm [18] to work with a polyhedron represented as a set of rays and vertices [19]. These two techniques have the good property of producing a code without any redundant control (for only one polyhedron), but while the second one can generates a very compact code, the first one can quickly explode in length.

The problem of scanning more than one polyhedron in the same code was firstly solved by generating a naive perfectly nested code and then (partially) eliminating redundant guards [15]. Another way was to generate the code for each polyhedron separately, and then to merge them [14, 5]. This solution generates a lot of redundant control, even if there were no redundancies in the separated code. Quilleré et al. proposed to recursively generate a set of loop nests scanning several unions of polyhedra by separating them into subsets of disjoint polyhedra and generating the corresponding loop nests from the outermost to the innermost levels [21]. This later approach provides at present the best solutions since it guarantees that there is no redundant control. However, it suffers from some limitations, e.g. high complexity or needless code explosion. The present work presents some solutions to these drawbacks.

7. Conclusion

The current trend in program optimization is to separate the selection of an optimizing transformation and its application to the source code. Most transformations are reorderings, followed optionally by modifications to the statements themselves. The program transformer must be informed of the selected reordering, and this is usually done by way of directives, like *tile* or *fuse* or *skew*. It is difficult to decide the completeness of a set of directives, or to understand their interactions. We claim that giving a scattering function is another way of specifying a reordering, and that it has several advantages over the directive method. It is more precise, it has better compositionality properties, and there are many cases in which automatic selection of scattering functions is possible. This paper provides for this purpose a flexible transformation framework for state-of-the-art parallelization and optimization techniques, by removing any additional constraint to the scattering function affinity. The only drawback was that deducing a program from a scattering function took time, and was likely to introduce much runtime overhead. We believe that tools like CLooG have removed these difficulties. The whole source-to-polyhedra-to-source transformation was successfully applied to the 12 benchmarks with a significant speedup of 4.05 with respect to the most widely used code generator, for the benchmark parts it is able to deal with.

Ongoing work aims at reasoning upstream from code generation step. Pointing out the most compute intensive parts in the source programs [6] would allow to drive the code generator to avoid meaningless, time and code size consuming control overhead removing. Another way to reduce both complexity and code versioning is to find the affine constraints on and between every static control part parameters [8].

Acknowledgments

The author would like to thank Paul Feautrier for his help in writing this paper. Many thanks also to all the CLooG's contributors and especially to Sven Verdoolaege.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.

[2] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubljana, october 2003.

[3] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, october 2003.

[4] Y. Bouchebaba. *Optimisation des transferts de données pour le traitement du signal: pavage, fusion et réallocation des tableaux*. PhD thesis, École des mines de Paris, 2002.

[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] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *Int. Conference on Supercomputing*, pages 278–285, Philadelphia, may 1996.

[7] J.-F. Collard, T. Risset, and P. Feautrier. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, 1995.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Jan. 1978.

[9] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, 1994.

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

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

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

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

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

[15] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

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

[17] M. Le Fur. Parcours de polyèdres paramétrés avec l'élimination de Fourier-Motzkin. Technical Report 2358, INRIA, 1994.

[18] H. Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[19] H. Le Verge, V. Van Dongen, and D. Wilde. Loop nest synthesis using the polyhedral library. Technical Report 830, IRISA, 1994.

[20] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.

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

[22] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.

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

[24] Y. Slama and M. Jemni. Vers l'extension du modèle polyédrique aux transformations irrégulières. In *CARI'5 International Conference on African Research in Computer Science*, Antananarivo, october 2000.

[25] S. Wetzel. Automatic code generation in the polytope model. Master's thesis, Facultät für Mathematik und Informatik, Universität Passau, 1995.

[26] D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.

[27] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[28] J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.