Efficient Code Generation for Automatic Parallelization and Optimization

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

Abstract

Supercompilers look for the best execution order of the statement instances in the most compute intensive kernels. It has been extensively shown that the polyhedral model provides convenient abstractions to find and perform the useful program transformations. Nevertheless, the current polyhedral code generation algorithms lack for flexibility by adressing mainly unimodular or at least invertible transformation functions. Moreover, their complexity is challenging for large problems (with many statements). In this paper, we discuss a general transformation framework able to deal with non-unimodular, non-invertible functions. A completed and improved version of one of the best algorithms known so far is presented to actually perform the code generation. Experimental evidence proves the ability of our framework to handle real-life problems.

I Introduction

The focus of recent compiler research is optimized code generation. The reason is that, with a higher level of integration, modern processors have acquired high-level features (vector processing, hidden and apparent parallelism, memory hierarchies), which are not available in so-called high-level languages, which were patterned after the much simpler processors of the 1960's. Hence, there is a semantic gap which grows larger and larger with time. A striking example is the Multimedia Instruction Set of many popular processors, which so far cannot be used directly in any high-level language.

Simple optimizations, like constant folding, can be directly applied on the abstract syntax tree of the program, but the most interesting ones (e.g. loop inversion) incur modifying the execution order of the program, and this has nothing to do with syntax. In many cases, optimization is done in three steps: (1) select a new execution order, most of the time the result is not a program, but a reordering function (a schedule, or a placement, or a chunking function); (2) build a loop nest (or a set of loop nests) which implement the execution order implied by the reordering function; (3) apply the local optimizations which have been enabled by the new execution order: for instance, mark some loops for parallel execution.

The main theme of this paper is the second step of the above scheme. Finding suitable execution orders has been the subject of most of the research on the *polytope model* [5, 9, 14, 16, 19] (an exhaustive related work can be found in the long version of the paper [2]) and the post-processing is easy in general. On the other hand, simple-minded schemes for loop building have a tendency to generate inefficient code, which may offset the optimization they are enabling. In this paper, I will show that, starting from one of the best algorithms known so far [15], one can generalize and improve it in two directions:

- The quality of the target code is such that many loop nests in familiar benchmarks can be regenerated optimally.
- The implementation is efficient enough to handle problems with thousands of statements and tens of free parameters.

The paper is organized as follows. Section II introduces some relevant definitions and shows the linear algebraic representation that can be used for describing parts of a program in optimizing or parallelizing compilers. Section III presents the polyhedra scanning problem and in particular a general program transformation framework in the polyhedral model. Section IV describes the code generation algorithm and proposes new ways to achieve an efficient target code. In section V, experimental results obtained through the algorithm implementation are shown. Finally, section VI summarizes the main contributions of this paper then discusses future works.

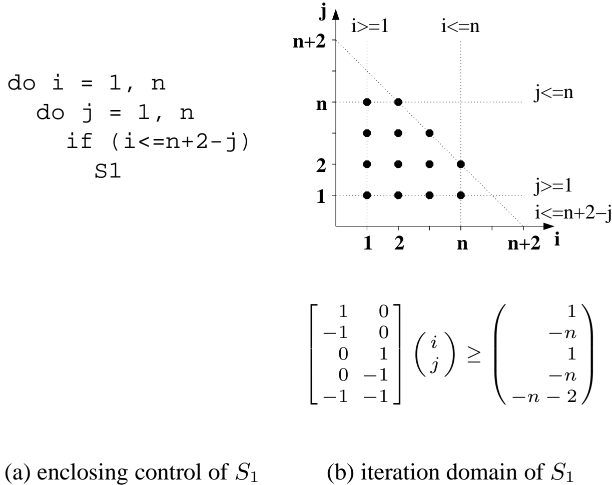
II Background and Notations

The loops in every imperative languages like C or FORTRAN can be represented using a n-entry column vector

called its *iteration vector*:

$$\vec{x} = (i_1, i_2, \ldots, i_n)^T,$$

where i_k is the kth loop index and n is the innermost loop. The surrounding loops and conditionals of a statement define its *iteration domain*, i.e. for which values of the iteration vector the statement has to be executed. When loop bounds and conditionals only depend on surrounding loop counters, formal parameters and constants, the iteration domain can always be specified by a set of linear inequalities defining a polyhedron [12]. 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 [17]. A maximal set of consecutive statements in a program with such polyhedral iteration domains is called a *static control part* (SCoP). The figure 1 illustrates the correspondence between static control and polyhedral domains.



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

(a) enclosing control of S_1 (b) iteration domain of S_1

Figure 1. Static control example

The execution of the statement instances follows *lexicographic order*. This means in a n-dimensional polyhedron, the operation corresponding to the integral point defined by the coordinates $(a_1...a_n)$ is executed before those corresponding to the coordinates $(b_1...b_n)$ iff

$$\exists i, 1 \leq i < n, (a_1...a_i) = (b_1...b_i) \land a_{i+1} < b_{i+1}.$$

III Polyhedra Scanning Problem

In the polyhedral model, code generation amounts to a *polyhedra scanning problem*. We consider it as a two-part specification for each statement

- A *union of disjoint polyhedra*, each polyhedron being defined by a set of parameterized inequalities in the n-dimensional space:

  $$\mathcal{D}(\vec{p}) = \left\{ \vec{x} \mid \vec{x} \in \mathbb{Z}^n, A\vec{x} + A'\vec{p} + \vec{c} \geq \vec{0} \right\},$$

  where $\vec{x}$ is the iteration vector, A and A' are constant matrices, $\vec{p}$ represents the parameters, and $\vec{c}$ is a constant vector. Considering a statement in a depth-n loop nest, the union of disjoint polyhedra defines the space where the iteration domain is included. As regards the vector $\vec{p}$, it is a merging of the structure parameters of the SCoP, i.e. the symbolic constants, mostly array sizes or iteration bounds.

- A *scheduling function*, an affine function specifying a scanning order for the integral points belonging to the union of iteration domains:

  $$\theta(\vec{x}) = T\vec{x} + T'\vec{p} + \vec{t},$$

  where T and T' are constant matrices, and $\vec{t}$ is a constant vector. Depending on the context, the scanning function may have several interpretations: to distribute the iterations across different processors, to order them in time, or both (by composition), etc.

The problem boils down to finding a set of nested loops visiting each integral point, once and only once, following the scanning order. This is a generalization of the usual polyhedra scanning problem since, to the author's knowledge, all previous works addressed the problem of scanning polyhedra in their *lexicographic* order. These approaches are inherently limited since they do not benefit from the opportunities for optimization that arise when the scanning order is only partially constrained. To actually perform a program transformation, we consider the original iteration domains of the statements as polyhedra with their original scanning order, i.e. the lexicographic order. We then apply the transformation by modifying their scanning order and we eventually generate the target code. A useful analogy is linear algebra: it is sometimes easier to change the basis to make some calculations and then to return back to the original basis.

Previous work on code generation in the polyhedral model required severe limitations on the transformation functions, e.g. to be unimodular [1, 13] (the T matrix has to be square and has determinant ± 1) or at least to be invertible [14, 19, 16, 5]. The underlying reason was, considering an original polyhedron defined by $A\vec{x} \geq -\vec{c}$ and the transformation function T leading to the target index $\vec{y} = T\vec{x}$, to find the transformed code is equivalent to scanning the polyhedron defined by $(AT^{-1})\vec{y} \geq -\vec{c}$.

We do not lay down any constraint on the transformation functions in particular because we do not try to perform a

$$\mathcal{T}(\vec{p}) = \left\{ \left(\frac{\vec{y}}{\vec{x}} \right) \middle| \left[\begin{array}{c|c} Id & -T \\ \hline 0 & A \end{array} \right] \left(\frac{\vec{y}}{\vec{x}} \right) + \left[\frac{-T'}{A'} \right] \vec{p} + \left(\frac{-\vec{t}}{\vec{c}} \right) \frac{=}{\geq} \vec{0} \right\} \quad (1)$$

change of basis of the original polyhedron to the target index. Instead, we apply a new scanning order to the polyhedra by adding new dimensions in leading positions, i.e. in order to change the lexicographic order. Thus, from each polyhedron $\mathcal{D}(\vec{p})$ and scheduling function θ, we build another polyhedron $\mathcal{T}(\vec{p})$ with the desired lexicographic order as shown in formula 1, where by definition, $(\vec{y}, \vec{x}) \in \mathcal{T}(\vec{p})$ if and only if $\vec{y} = \theta(\vec{x})$. The new polyhedron has to be scanned lexicographically until the last dimension of $\vec{y}$. Then there is no particular order to respect for the remaining dimensions.

By using such a transformation policy, the data of both original iteration domain and transformation are included in the new polyhedron. The additional dimensions do not change the number of integral points but carry the transformation data. This is helpful since we have to update the references to the iterators in the loop body, and *necessary* when the transformation is not invertible. Moreover, the additional dimensions carry the loop strides (the step of the iterator after one iteration); this will be detailed in section IV-B.

IV Extended Quilleré et al. Method

Once the transformations have been applied to the original polyhedra, it remains to generate an efficient scanning code. 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. Both valuations are contradictory, but the simplest solutions lead to a large amount of redundant control, especially when more than one polyhedron is considered.

At present, the Quilleré et al. method give the best results when we have to generate a scanning code for several polyhedra [15]. This technique is guaranteed to avoid redundant control while scanning the scheduled dimensions. However, it suffers from some limitations, e.g. high complexity, code generation with unit strides only and unoptimized code when the scheduling only partially constrains the scanning order. In the following, we propose some solutions to these drawbacks. We present the general algorithm with some adaptations to our purpose in section IV-A. We address the non-unit strides problem in section IV-B. Finally in section IV-C we show how it is possible to benefit from a relaxed scheduling.

IV-A Code Generation Algorithm

The main part of the algorithm is a recursive generation of the scanning code, maintaining a list of polyhedra from the outermost to the innermost loops. It makes a strong use of polyhedral operations that can be achieved by e.g. PolyLib[1] [18]. The basic idea is to generate loop levels by projecting the polyhedra onto the corresponding dimension. Then by splitting the projections into disjoint polyhedra and sorting the resulting polyhedra in order to respect the lexicographic order. Lastly, by recursively generating the loop nests that scan each polyhedron for the next dimensions. The algorithm is described in figure 2. The considered polyhedra $\mathcal{D}$ are not only a set of constraints: they may carry either a statement body S_i (notation $\mathcal{D}^{S_i}$) or a polyhedra list (notation $\mathcal{D}^i \rightarrow (...)$). The algorithm is started with 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 $d = 1$.

This algorithm is slightly different from the one presented by Quilleré et al. in [15]; our two main contributions are the following: the support for non-unit strides (step 5a, see section IV-B) and the exploitation of freedom degrees (i.e. when some operations do not have a complete schedule) to produce a more effective code (step 5c, see section IV-C).

Let us describe this algorithm with a non-trivial example. We propose to scan the two polyhedral domains presented in Figure 3(a). The iteration vector is (i, j, k) and the parameter vector is (n). We first compute the intersections with the context (i.e., at this point, the constraints on the parameters, supposed to be $n \geq 6$). We project the polyhedra onto the first dimension, i, then we separate them into disjoint polyhedra. This means that we compute the domains where there are points to scan for $\mathcal{T}^{S_1}$ alone, both $\mathcal{T}^{S_1}$ and $\mathcal{T}^{S_2}$, and $\mathcal{T}^{S_2}$ alone (as shown in Figure 3(b), this last domain is empty). Here, we notice there is a constraint on an inner dimension 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 polyhedron 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 process is triv-

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

CODEGENERATION: Build a polyhedra scanning code without redundant control.

Input: a polyhedron list `list`, a context `C`, the current dimension `d`.

Output: the writing of the scanning code for the polyhedra inside `list`.

1. Intersect each polyhedron $\mathcal{T}^{S_i}$ in the list with C in order to restrict the scanning code to the context of its own loop nest: `list := intersection(list,C)`
2. Compute for each polyhedron $\mathcal{T}^{S_i}$ its projection $\mathcal{P}^i$ onto the outermost `d` dimensions and consider the new list of $\mathcal{P}^i \rightarrow \mathcal{T}^{S_i}$: `list := projection(list,d)`
3. Separate the projections into a new list of disjoint polyhedra: given a list of n polyhedra, we have to compute $(\mathcal{P}^1 - \mathcal{P}^2) \rightarrow \mathcal{T}^{S_1}$, $(\mathcal{P}^1 \cap \mathcal{P}^2) \rightarrow (\mathcal{T}^{S_1}, \mathcal{T}^{S_2})$ and $(\mathcal{P}^2 - \mathcal{P}^1) \rightarrow \mathcal{T}^{S_2}$, then to do the same for each resulting polyhedron with $\mathcal{P}^3 \rightarrow \mathcal{T}^{S_3}$, etc.: `list := separate(list)`
4. Sort the list such that a polyhedron is before another one if its scanning code has to precede the other to respect the lexicographic order: `list := sort(list)`
5. For each polyhedron $\mathcal{P} \rightarrow (\mathcal{T}^{S_m}, ..., \mathcal{T}^{S_n})$ in the list: `For each polyhedron in list do`
   (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 $(\mathcal{T}^{S_m}, ..., \mathcal{T}^{S_n})$ list (see section IV-B):
   `stride = find_stride(polyhedron)`
   (b) Write the scanning informations for the dimension d (i.e. guards, loop lower and upper bounds) from the constraints in the polyhedron $\mathcal{P}$ simplified in the context of C:
   `write_scanning_code(polyhedron,C,stride,d)`
   (c) While there is a polyhedron in $(\mathcal{T}^{S_m}, ..., \mathcal{T}^{S_n})$:
   `while (inner_polyhedron in inner_list)`
      i. Merge successive polyhedra with another dimension to scan in a new list:
      `new_list := merging_non_terminal(inner_list)`
      ii. Recurse for the next dimension $d+1$ with the new loop context $C \cap \mathcal{P}$:
      `CodeGeneration(new_list,intersection(polyhedron,C),d+1)`
      iii. If there is a $\mathcal{T}^{S_i}$ in the list, write the statement body S_i:
      `if (inner_polyhedron) write_body(inner_polyhedron)`
   (d) Close the open braces of the loop: `write_end_scanning_code(d)`

Figure 2. Extended Quilleré et al. Algorithm

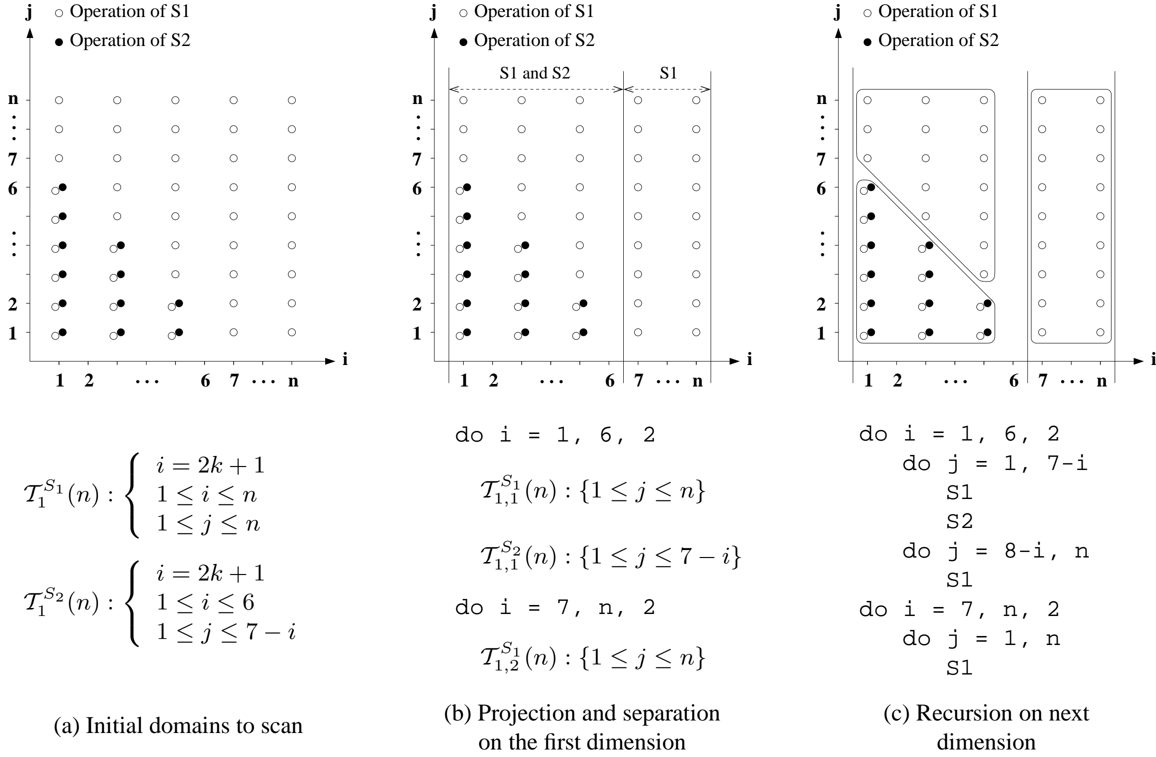
ial for the second list but yields several domains for the first list, as shown in Figure 3(c). Eventually, we generate the code associated with the new dimension, and since this is the last one, the scanning code is fully generated.

IV-B Non-Unit Strides

To scan a transformed polyhedron it may be necessary to avoid some values in some dimensions. This happens when there exists a set of dimensions such that the transformed polyhedron projection onto these dimensions has integer points without a corresponding image in the original space.

Previous works were challenged by this problem, which occurs when the transformation function is non-unimodular (i.e. the transformation matrix T has non unit determinant). We can observe the phenomenon with the transformation of the polyhedron in figure 4(a) by the function $\theta(i) = i + 2j$ (the corresponding transformation matrix $T = \left[\begin{array}{cc} 1 & 2 \end{array}\right]$ is not invertible, but it can be extended to $T = \left[\begin{array}{cc} 1 & 2 \\ 1 & 0 \end{array}\right]$). The target polyhedron is shown in figure 4(a). The integer points without heavy dots have no images in the original polyhedron. The original coordinates can be determined from the target ones by $\vec{original} = T^{-1}\vec{target}$. 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 scanning the holes, the loop strides (the steps the iterators make at the end of the loop body) and the loop lower bounds had to be found. Many works proposed to use the Hermite Normal Form [17] in different ways to solve the problem.

In this paper we do not change the basis of the original polyhedra, but we only change their scanning order as discussed in section III. As a consequence, our target sys-



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

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

(a) Initial domains to scan

$\mathcal{T}_{1,1}^{S_1}(n) : \{1 \le j \le n\}$

$\mathcal{T}_{1,1}^{S_2}(n) : \{1 \le j \le 7-i\}$

$\mathcal{T}_{1,2}^{S_1}(n) : \{1 \le j \le n\}$

(b) Projection and separation on the first dimension

(c) Recursion on next dimension

Figure 3. Step by step code generation example

tems are always integral and there are no holes in the corresponding polyhedra. As an illustration, the target polyhedron given by our transformation policy is shown in figure 4(c). The stride informations are explicitly contained in the constraint systems thanks to the equalities. Without the step 5a in the code generation algorithm, the successive projections lead to the equality constraints, as shown in figure 5(a) for our example. This leads to an inefficient scanning code since heavy guards (with modulo operations) are inside the loops.

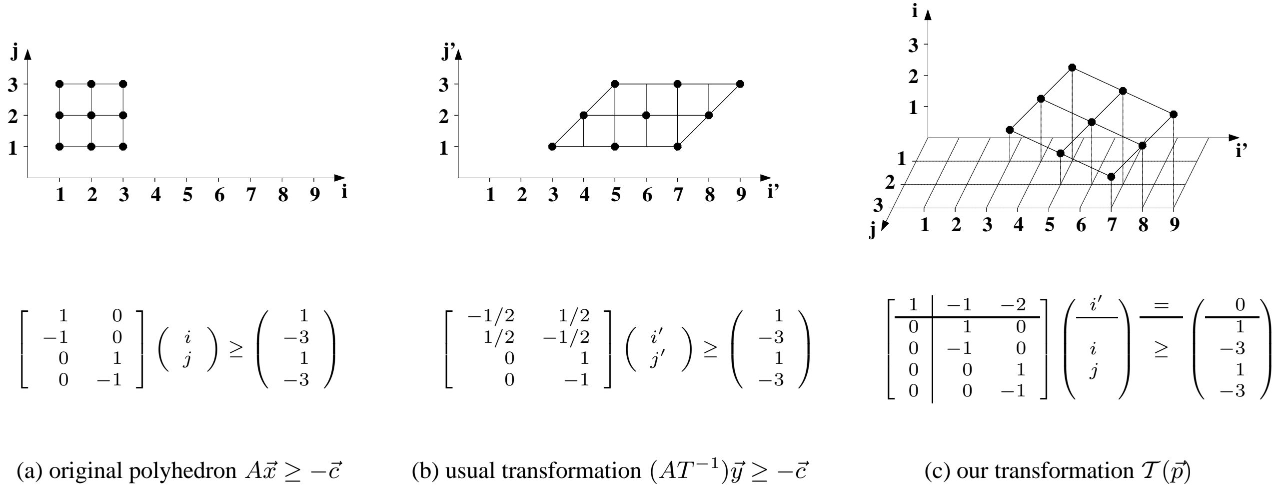
We solve this problem by finding the stride for the current dimension x and the new lower bound, i.e. the first value in the dimension with an integral point to scan (for instance if we want to scan only odd or even points onto a dimension, the stride is the same but we need to start the loop from a convenient odd or even point). The stride can be directly read from the equality constraints $y^S = s^S x + n^S$ of the transformed polyhedra, where for the polyhedron S, y^S is the striding dimension, s^S is the stride, x is the strided dimension, and n^S is a merging of the parameters (outer dimensions, structure parameters and constant). The lower bound has to be found under the context of the outermost loop counters and the structure parameters. This is a problem in parametric integer programming, that can be solved by PipLib[2] [8]. For our example, the stride of the i loop is 2, directly given by the equality constraint, and the lower bound is the minimum value of i in the polyhedron defined by $\begin{cases} 1 \le i \\ i' - 6 \le i \\ i = i' - 2j \end{cases}$ and under the context $3 \le i' \le 9$. The solution given by PipLib as a quasi affine selection tree is shown in the optimized final code in figure 5(b).

When there is a set of polyhedra K to scan, for each dimension x we have to consider a set of striding constraints $y^S = s^S x + n^S$. Like in [11], the greatest common step gcs is given by:

$$gcs = gcd(\{s^{S_i} | S_i \in K\}, \{n^{S_i} - n^{S_j} | S_i, S_j \in K \wedge i \neq j\}).$$

We then have to find the lower bound, i.e. the first value of x where an integer point has to be scanned. This can be achieved in the same way as for one polyhedron by merging all the constraints of the different polyhedra for the dimension x. We have successfully implemented this method with the restriction that the n^S has to be constant. The problem to find the gcs when the n^S are parameterized is under investigation.

[2]PipLib is available at `http://www.prism.uvsq.fr/~cedb`



(a) original polyhedron $A\vec{x} \geq -\vec{c}$ (b) usual transformation $(AT^{-1})\vec{y} \geq -\vec{c}$ (c) our transformation $\mathcal{T}(\vec{p})$

Figure 4. Non-unimodular transformation with $\theta(i) = i + 2j$

```
do i'=3, 9
  do i=MAX(1,i'-6), MIN(i'-2,3)
    if (MOD(i'-i,2) == 0) then
      j=(i'-i)/2
      S1
```

(a) guarded version

```
do i'=3, 9
  if (i'<=7)
    lower = i' - 2*((i'+1)/2) + 2
  else
    lower = i' - 6

  do i=lower, MIN(i'-2,3), 2
    j = (i'-i)/2
    S1
```

(b) strided version

Figure 5. Scanning codes for the polyhedron in figure 4(c)

IV-C Exploitation of Freedom Degrees

The polyhedron scanning orders specified by the scheduling functions may leave some dimensions unspecified. This means that the code generator is free to choose their scanning order. Basically, this can happen when the operations are parallel onto these dimensions, and when there is no dependences between the considered statements. Then it is the code generator responsibility to provide the best target code i.e. with the minimum control overhead. This work is essential since it concerns the innermost loops of the generated code, where the consequences of a bad control management are the most disturbing. For instance let us consider the matrix multiply codes in figure 6. These target codes can result from a generation where the scheduling functions are $\theta^{S_1}(\vec{x}) = \vec{t}$ and $\theta^{S_2}(\vec{x}) = \vec{t}$ (this is possible since the computations on remaining dimensions are fully parallel).

```
do t=1, n
  do i=1, n
    do j=1, n
      c1(i,j)+=a1(i,t)*b1(t,j)
  do i=1, n
    do j=1, n
      c2(i,j)+=a2(i,t)*b2(t,j)
```

(a) splitted version (iterations: $2n^3 + 2n^2 + n$)

```
do t=1, n
  do i=1, n
    do j=1, n
      c1(i,j)+=a1(i,t)*b1(t,j)
      c2(i,j)+=a2(i,t)*b2(t,j)
```

(b) merged version (iterations: $n^3 + n^2 + n$)

Figure 6. Equivalent target codes for matrix multiplies

We have tested these simple codes on a x86 architecture at 1 GHz, compiled with the GCC 3.2.2 compiler and the option -O3. For $n = 500$ the code in figure 6(a) takes $5.31s$ while the code in figure 6(b) takes $4.75s$, a 15% improvement. The code in figure 6(c) can be generated by performing the Quilleré et al. recursion for every free dimensions of the polyhedra. After each recursion, if some polyhedra are fully scanned, the corresponding statement bodies have to

be printed out. A new list with the remaining polyhedra is created to continue the recursion. This is the aim of step 5c of the algorithm. Unfortunately this solution is only partial, since it allows us to reduce the control overhead only according to the original lexicographic order.

The compulsory control for a polyhedra scanning problem without scheduling constraints is the number of iterations necessary to scan the biggest polyhedra in term of integral points. The control is minimum when it is limited to this compulsory control. We are free to modify the scanning orders to achieve this goal. Formally, the general problem is to find for each polyhedron defined by $A^S\vec{x} \geq -\vec{c^S}$ an invertible transformation matrix Z^S and a translation vector $\vec{z^S}$ such that the number of integral points in the intersection of all the polyhedra $\cap_S(A^S Z^S \vec{x} \geq \vec{z^S} - \vec{c^S})$ is maximum. Because of the Z^S matrices, the general problem is not affine. Then we simplify it by considering only the translation vector $\vec{z^S}$.

Counting integer points inside a parameterized polyhedron is possible using Ehrhart polynomials [6]. These polynomials can have periodic coefficients when a vertex of the counted polyhedron is not integral. In a code generation framework this case is rare, but when this happens we are not able to use this method for our purpose. Once the Ehrhart polynomials are calculated, it is not hard to find for which translation the maximum number of integral point in the intersection is achieved. Then we can apply the translation to the corresponding polyhedra and generate a scanning code optimized in control. This technique is guaranteed to reduce the control overhead, at worst, it will leave the original polyhedra intact. Using Z^S to find new solutions to the polyhedron overlapping problem is left for future works.

V Experimental Results

Our implementation of this algorithm is called CLooG[3] (Chunky Loop Generator) and was originally designed for a locality-improvement algorithm and software (Chunky) [4]. Thanks to an implementation of a SCoP extraction algorithm into Open64 [3], a study on the applicability of the presented framework to several benchmarks has been achieved. The chosen methodology was to perform the code regeneration of all static control parts of a representative set of benchmarks.

Figure 7 summarizes the results for a set of SpecFP 2000 and PerfectClub benchmarks. The first three columns shows some general informations about the SCoPs: the first one gives the total number of SCoPs in the corresponding benchmark, the next two columns give some precisions about how many of them are *rich*, i.e. enclose at least one loop, and count the number of *rich* SCoPs with at least one global parameter. In brackets is shown the maximum number of parameters among the SCoPs. Not surprisingly, the set of problems appears to be heavily parametric, supporting the works on fully parametric methods, but challenging the code generators since free parameters are the main source of memory explosions. The *Iteration Domains* section describes the shape of the polyhedra: a *point* means that the corresponding statement is executed only once, a *rectangle* is an iteration domain bounded by constants, a *prism* is bounded by constants except for one bound, and an *other* has more than one varying bound. Lastly, the *Code Generation* section describes the code generator's behavior on a x86 architecture at 1 GHz with 256 MB RAM. The first column shows how many SCoPs have to be regenerated in a suboptimal way because of a memory explosion on the testing system. The three challenging problems have the common property to be heavily parametric (13 or 14 free parameters). The *Duplication* column shows the duplication factor between original and target codes, it appears to not be very high (3.4 for the whole benchmark set), The *Time* column shows the time spent during the code generation processing.

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. Previously related experiences with Omega [11] or LooPo [10] showed how it was challenging to producing efficient code just for ten or so polyhedra whithout time or memory explosion.

VI Conclusion

The complexity of code generation has long been a deterrent for using polyhedral representations in optimizing or parallelizing compilers. Moreover, most existing solutions only address a subset of the possible polyhedral transformations. The contribution of this paper is twofold. First, it presents a general transformation framework. This results in opening new opportunities to optimize the target program, e.g. to benefit from more freedom while generating the code. Efficient solutions have been proposed to use them. Second, it demonstrates the ability of a code generator to produce an optimal control on real-life problems, with a possibly very high statement number, in spite of a worst-case exponential complexity.

Ongoing work aims at finding new improvements on the target code quality. Two major challenges are to solve the general greatest common step problem for parameterized non-unit stride, and to find new answers to the polyhedra overlapping question. There are still challenging problems leading to time or memory explosion. Pattern matching, i.e.

[3]CLooG is available at `http://www.prism.uvsq.fr/~cedb`

	SCoPs			Iteration Domains					Code Generation		
	All	Rich	Parametric	All	Point	Rectangle	Prism	Other	Suboptimal	Duplication	Time (s)
applu	25	19	15(6)	757	233	506	4	2	0	1.5	32
apsi	109	80	80(14)	2192	1156	1036	0	0	1	4.1	58
art	62	28	27(8)	499	331	142	0	0	0	1.9	2
lucas	11	4	4(13)	2070	317	1753	0	0	1	3.3	127
mgrid	12	12	12(4)	369	314	55	0	0	0	1.2	5
quake	40	20	14(7)	639	367	216	9	0	0	1.1	8
swim	6	6	6(3)	123	63	60	0	0	0	1	1
adm	109	80	80(14)	2260	1224	1036	0	0	1	4.1	59
dyfesm	112	75	70(4)	1497	880	540	33	1	0	1.3	18
mdg	33	17	17(6)	530	358	167	5	0	0	1.1	5
mg3d	63	39	39(11)	1442	561	856	0	0	0	1.2	21
qcd	74	30	23(8)	819	458	361	0	0	0	14.6	69

Figure 7. Coverage of static control parts in high-performance applications

to short-cut the general polyhedral calculations for simple cases (e.g. rectangular domains), seems to be a promising way to reduce the time spent in code generation. It has been shown in this paper that the main explosion factor is the number of free parameters, since the variability of parameter interactions leads to an exponential growth of the generated code. Upstream from code generation, it is possible for compilers to reduce both complexity and code duplication by finding linear relations among variables [7].

Acknowledgments

The author would like to thank Paul Feautrier and François Thomasset for their valuable help and suggestions. Many thanks also to Albert Cohen and Saurabh Sharma for having made possible the experiments on benchmark sets.

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 (long version). Technical Report 2003/43, PRiSM, Versailles University, october 2003.

[3] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral transformations to work. Technical Report 4902, INRIA, july 2003.

[4] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 Int. Conf. on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, april 2003.

[5] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. volume 24, pages 421–444, 1998.

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

[7] 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.

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

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

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

[11] 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.

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

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

[14] 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.

[15] 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.

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

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

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

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