Generation Of Efficient Nested Loops From Polyhedra

Fabien Quilleré, Sanjay Rajopadhye
Irisa, Campus de Beaulieu, Rennes, France

Doran Wilde
Brigham Young University, Provo, Ut, USA

Abstract. Automatic parallelization in the polyhedral model is based on affine transformations from an original computation domain (iteration space) to a target space-time domain, often with a different transformation for each variable. Code generation is an often ignored step in this process that has a significant impact on the quality of the final code. It involves making a trade-off between code size and control code simplification/optimization. Previous methods of doing code generation are based on loop splitting, however they have non-optimal behavior when working on parameterized programs. We present a general parameterized method for code generation based on dual representation of polyhedra. Our algorithm uses a simple recursion on the dimensions of the domains, and enables fine control over the tradeoff between code size and control overhead.

Keywords: Automatic Parallelization, Polyhedral Model, Nested Loops, Code Generation

1. Introduction

The polyhedral model is a formalism for reasoning about parallel computations, using *Systems of Affine Recurrence Equations* defined over polyhedral shaped domains. Its use in automatic parallelization of nested loops goes back to the work of Kuck (1978), who showed that the domain of nested loops with affine lower and upper bounds can be described in terms of a polyhedron, and the seminal work of Karp Miller and Winograd (1967) on scheduling systems of uniform recurrence equations. The model draws on a number of results from data flow analysis of static loop programs (Feautrier, 1991), scheduling, using compact representations, of indexed computations (essentially uniform or affine recurrence equations) in the context of parallelization (Feautrier, 1992; Karp et al., 1967) or systolic array synthesis (Rajopadhye and Fujimoto, 1990; Quinton and Van Dongen, 1989). Independently of the polyhedral model, many authors have developed linear-algebraic formulations for a large number of loop transformations (Banerjee, 1993; Irigoin, 1988; Wolf and Lam, 1991), which consist of reindexing the source index domain using affine transformations and then scanning the new domain, generally in a different order.

There has been considerable work on finding good transformations to optimize various measures of performance such as time, as in the work on scheduling (Feautrier, 1992; Darte and Vivien, 1996) communication volume (Darte and Robert, 1993; Feautrier, 1994), synchronizations, etc. However, code generation, the back-end of automatic parallelization, has received relatively less attention, even though it has a significant impact on the quality of the resulting code.

In this paper, we address the problem of generating imperative code for a set of statements defined over polyhedral context domains. The context domain of a statement is a finite union of (disjoint) polyhedral sets of index vectors, for which the statement has to be evaluated. The main idea is that since each domain is a union of disjoint polyhedra, it can be individually scanned by using a set of perfectly nested loops, which are then arranged in an appropriate textual order. This uses and generalizes the techniques for the polyhedron scanning problem, namely finding a set of nested `do` loops which visit each integral point of a polyhedron in the lexicographic order. This problem was first studied by Ancourt and Irigoin (1991) and subsequently by many authors.

The loop body is the code implementing the evaluation of the statements. Generating code for statements (the loop body) is relatively straight forward and is not the focus of this paper. Instead, we focus on generating the loops and control code.

The code generation problem is more difficult when the domains are not disjoint. Furthermore, in the most general case, the domain of *each* statement may be a finite union of polyhedra. Typically, this necessitates imperfectly nested loops that enumerate all the index vectors of the domain.

We present an algorithm to scan a union of polyhedral domains in the lexicographic order of their indices. The method can deal with affine-by-statement and even piecewise-affine mappings, since domains can be separated into multiple sub-domains, one for each mapping.

The remainder of this paper is organized as follows. In Sect. 2, we first present notations and background material, including details of some of the routines we use for manipulating polyhedra. These routines are drawn from a library, Polylib, developed at Irisa (Wilde, 1993). Polylib is freely available under Gnu license[1]. Next (in Sect. 3), we discuss the problem of scanning a single polyhedron and recall the results of Le Verge et al. (1994), which exploit the dual representation used in Polylib. In Sect. 4 we formulate and discuss the issues involved in scanning a union of (not necessarily disjoint) polyhedra. In Sect. 5 we present our algorithm, first as an overview using an illustrative example

[1] See `http://www.irisa.fr/cosi/Polylib`

and then in more detail. An important subproblem that arises is that of "sorting" polyhedra, which is also discussed there. In Sect. 6 we develop a number of extensions in notably to (i) tradeoffs between code-size and efficiency, (ii) non-unimodular mapping, and (iii) generation of (data) parallel code. Finally, we discuss related work in Sect. 7, and present our conclusions (Sect. 8).

2. Notation and Background

We now introduce the basic mathematical notation and also the tools from the Polylib library that we need to manipulate polyhedral domains. Polylib was first developed at Irisa (Wilde, 1993), and later extended by a number of authors, notably at Brigham Young University and Strasbourg.

2.1. Basic Notation

A statement S defined over a polyhedral domain $\mathcal{D}$ is called a domain-qualified statement, and we denote it $\mathcal{D} :: S$. A system of recurrence equations can always be rewritten as a set of domain-qualified statements, as shown in Fig. 1.

In our notation, upper case indices are program parameters and are not explicitly declared in the indices of a domain. As explained later, this is only a matter of notation. In the polyhedral model, the symbolic program parameters are represented by dimensions added to the polyhedral domain of each statement. In the example of Fig. 1, the domain of statement `S1` may be viewed either as a 2-dimensional parameterized domain or equivalently, as a single 4-dimensional polyhedron.

In this representation, the set of constraints that can be expressed on the parameters is the standard one of the polyhedral model, i.e., linear constraints. The constraints on the symbolic parameters can be specified as a polyhedron, whose dimension is the number of parameters.

We now review the concept of polyhedra to help in the understanding of the polyhedron scanning problem. A polyhedron is defined by $\mathcal{D} = \{ \ x \mid Ax \geq b \ \}$ where A and b are a constant matrix and vector respectively. A parameterized polyhedron is a family $\mathcal{D}(p)$ of polyhedra, one per instance of the parameters p, and may be described by replacing vector b with an affine combination of the parameters:

$$\mathcal{D}(p) = \{ \ x \mid Ax \geq Bp + b \ \}$$

where A and B are constant matrices and b is a constant vector. This parameterized polyhedron can be rewritten in the form of a non-

(a) Statements qualified by domains

```
{i,j | 1<=i<=N; 1<=j<=M} :: S1;
{i,j | i=j; 3<=j<=N} :: S2;
```

(b) View of the domains for the case where $4 \leq M \leq N$

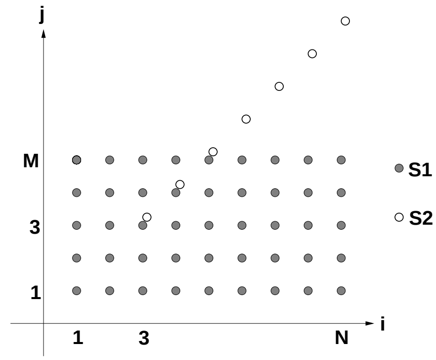


Figure 1. Systems of statements qualified by domains

parameterized polyhedron in the combined data and parameter index space as:

$$\mathcal{D}(p) = \left\{ x \mid \begin{pmatrix} A & -B \end{pmatrix} \begin{pmatrix} x \\ p \end{pmatrix} \geq b \right\}$$

$$\mathcal{D}' = \left\{ \begin{pmatrix} x \\ p \end{pmatrix} \mid A' \begin{pmatrix} x \\ p \end{pmatrix} \geq b \right\}$$

A polyhedron may be equivalently specified either with a set of constraints (inequalities) or by its generators (vertices and rays). This description was proposed by Motzkin (Motzkin et al., 1953). Polylib is based on this double description, and uses the Chernikova algorithm (Chernikova, 1965) to transform one to the other. It turns out to be very useful for many of the operations. However, keeping the dual representation is only feasible for small dimensional domains since a d-polyhedron with n constraints might have as many as $d^{\lfloor \frac{n}{2} \rfloor}$ rays/vertices, in the worst case. We rely on the fact that computational domains tend to be relatively small polyhedra (in terms of the dimension and number of constraints) due to the fact that loops are not deeply nested. We have not experienced significant problems, related to the space inefficiency.

2.2. Projection of polyhedra

The library function *DomainAddRays* joins a set of lines, rays, or points to a domain and produces the resulting domain with all redundancies eliminated. It is used in this paper to eliminate (or project out) the inner loop indices from the bound expressions of outer loops. This is

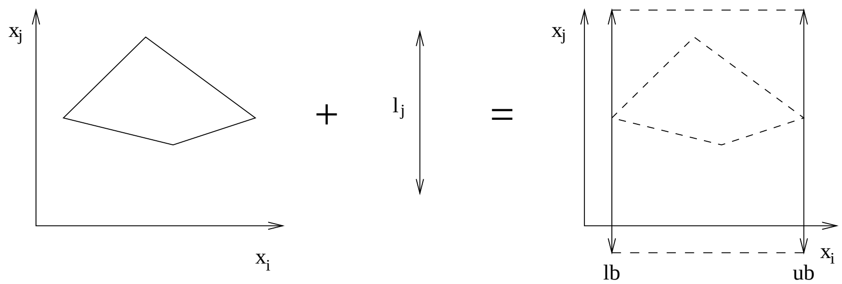


Figure 2. Adding a line to project out an index

illustrated in Fig. 2 where x_i is an outer loop variable and x_j is an inner loop variable. To compute the loop bounds for the x_i-loop as a function of parameters and outer loop variables, the inner loop variables x_j, for $j > i$ must be removed from the domain. This is done by projecting the domain in the direction of the inner loop variables onto the x_i-axis, giving lb and ub, the lower and upper bounds of loop variable x_i, respectively. The resulting polyhedron, as shown in the figure, is a cylinder, i.e., has no constraints on the inner loop variables.

Thanks to the dual representation, Polylib achieves this simply by adding lines $\{l_{i+1}, l_{i+2}, \cdots\}$ in the direction of all of the inner loop variables $\{x_{i+1}, x_{i+2}, \cdots\}$ to the vertex-ray representation of the polyhedron, and a call to the Chernikova algorithm.

2.3. Simplification of polyhedra in context

The simplification of a polyhedral domain $\mathcal{D}$ within a *context* domain $\mathcal{C}$ is defined as follows:

Given domains $\mathcal{D}$ and $\mathcal{C}$, then Simplify $(\mathcal{D}, \mathcal{C}) = \mathcal{D}'$, when

1. $\mathcal{D}' \cap \mathcal{C} = \mathcal{D} \cap \mathcal{C}$, $\mathcal{D}' \supseteq \mathcal{D}$, and
2. D' is maximal, in the sense that there does not exist $\mathcal{D}'' \supset \mathcal{D}'$ such that $\mathcal{P}'' \cap \mathcal{C} = \mathcal{D} \cap \mathcal{C}$

The simplify function finds the largest domain set (or domain with the smallest list of constraints) that, when intersected with the context $\mathcal{C}$ is equal to $\mathcal{D} \cap \mathcal{C}$. The simplify operation is done by computing the intersection $\mathcal{D} \cap \mathcal{C}$ and while doing so, recording which constraints of $\mathcal{D}$ are "redundant" with the result of the intersection. The result $\mathcal{D}'$ of the simplify operation is then the domain $\mathcal{D}$ with the "redundant" constraints removed. A simple example is shown in Fig. 3, where domain $\mathcal{D}$ is simplified (resulting in domain $\mathcal{D}'$) by eliminating the two constraints that are redundant with context domain $\mathcal{C}$.

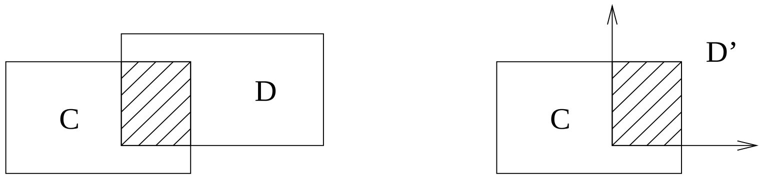


Figure 3. Domain Simplification: $\mathcal{D}' =$ Simplify $(\mathcal{D}, \mathcal{C})$

Polyhedral simplification will turn out to be extremely useful later when we seek to eliminate redundant guards and loop bounds tests.

3. Scanning a Single Polyhedron

In this section we describe a method to generate perfectly nested loops to scan a single polyhedron (Le Verge et al., 1994). Unlike the standard methods based on Fourier-Motzkin elimination (Ancourt and Irigoin, 1991), it exploits the dual representation of polyhedra, as implemented in Polylib.

We formulate the *polyhedron scanning problem* as follows: given a parameterized polyhedral domain $\mathcal{D}(p)$ in terms of a parameter vector p and a set of k constraints: $\mathcal{D}(p) = \{ \ x \mid Ax \geq Bp + b \ \}$, produce a set of loop bound expressions $lb_1, ub_1, \cdots lb_n, ub_n$ such that the loop nest:

$$\begin{array}{l} \mathtt{lb_1} \leq \mathtt{x_1} \leq \mathtt{ub_1} \\ \quad \mathtt{lb_2} \leq \mathtt{x_2} \leq \mathtt{ub_2} \\ \qquad \ddots \\ \qquad\quad \mathtt{lb_n} \leq \mathtt{x_n} \leq \mathtt{ub_n} \\ \qquad\qquad \text{loop body} \end{array}$$

will visit once and only once all integer points in the domain $\mathcal{D}(p)$ in lexicographic order of the elements of $x = (x_1, \cdots, x_n)$. For a loop variable x_i, we use the term *outer loops* to refer to loops which enclose the x_i-loop, that is, the loops over index variables x_j, for $j < i$. We use *inner loops* to refer to the loops contained in the x_i-loop, that is, the loops over indices x_j, for $j > i$. In the code to be produced, loop bounds should be the minimum or maximum of affine functions of outer loop variables and parameters.

The problem of finding loop bounds is related to integer linear programming and shares its complexity. Fortunately, these problems tend to be relatively small (in terms of the dimension and number of constraints) due to the fact that loops are not deeply nested, and exact solutions for typical problems can be found in reasonable time.

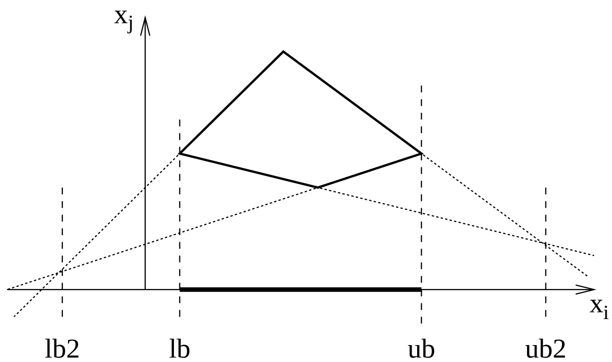


Figure 4. Projection using Fourier-Motzkin Elimination

3.1. Basis of the method

Our algorithm for scanning a single polyhedron resembles the standard method Fourier-Motzkin (FM) elimination (Ancourt and Irigoin, 1991), in that it projects the polyhedron in the canonical direction of inner loop variables in order to eliminate all constraints on them. Thus bounds on a loop are found independent of inner loop indices. However, the FM method considers all pairs of constraints when finding the bound on the projection. In Fig. 4, for example, lower bounds $lb2$ and lb and upper bounds $ub2$ and ub are all considered by the FM method. In general, given n constraints, as many as $\frac{n^2}{4}$ loop bounds could be generated in eliminating a single variable, and the total number grows exponentially with the number of variables. Most of these bounds turn out to be redundant (non-tight) and must be eliminated. The elimination of these redundant loop bounds is a significant problem when using the FM method.

Our algorithm only considers pairs of constraints that are *adjacent*, and therefore never generates any redundant bounds (like $lb2$ and $ub2$ in the example). Thus the redundant bound elimination phase of the FM method is replaced with an adjacency test on pairs of constraints in the proposed method. This test is accomplished very efficiently assuming the rays and vertices of the polyhedron are known. The algorithm maintains both the constraint and the ray/vertex representation of a polyhedron allowing the employ of the adjacency test.

3.2. Loop nest synthesis

We use a recursive algorithm which separates (or factors) a specified domain $\mathcal{D}$ into an intersection of the initial context domain $\mathcal{C}_0$ and a sequence of loop domains $\mathcal{D}_1,\ \mathcal{D}_2,\ \cdots$, such that:

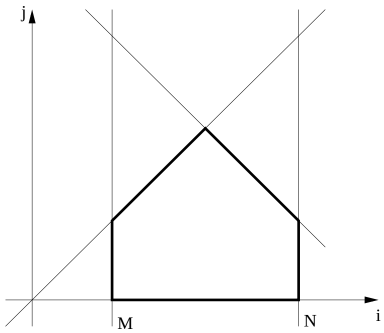


Figure 5. Domain `{i,j|j<=i<=N;j>=0;i>=M;i+j<=N+M;M>=2;M<N}`

1. $\mathcal{D} = \mathcal{C}_0 \cap \mathcal{D}_1 \cap \cdots \cap \mathcal{D}_n$

2. each loop domain $\mathcal{D}_i$ has no constraint involving inner loop variables $x_{i+1}, x_{i+2}, \ldots$, i.e., in these dimensions it is a cylinder. Thus, it is completely specified (in terms of constraints) by its projection, $\mathcal{L}_i$, on the outermost i dimensions. We sometimes abuse the notation and use $\mathcal{D}$ and $\mathcal{L}$ interchangeably.

3. the linear constraints defining each loop domain $\mathcal{D}_i$ are not redundant with any constraint established by any outer loop (i.e., within $\mathcal{C}_{i-1} = \mathcal{C}_0 \cap \mathcal{D}_1 \cap \cdots \cap \mathcal{D}_{i-1}$, which we call the *context* of this loop).

Our algorithm "splits" the constraints of $\mathcal{D}$ onto n separate (but not necessarily disjoint, remember that they have the same number of dimensions) polyhedra. The constraints defining each polyhedron $\mathcal{D}_i$ specify the lower and upper bounds of loop index i.

At each step i of the recursion, with a context $\mathcal{C}_{i-1}$, we generate one additional loop domain $\mathcal{D}_i$ by

- projecting out the inner loop variables x_j, for $j > i$, from domain $\mathcal{D}$;

- simplifying the resulting domain $\mathcal{D}_i$ in the context $\mathcal{C}_{i-1}$; and

- determining the context $\mathcal{C}_i = \mathcal{C}_{i-1} \cap \mathcal{D}_i$ for the next step of the recursion.

The decomposition of $\mathcal{D}$ into a sequence of loop domains $\mathcal{D}_i$ yields a loop nest where each loop of depth i is specified by the polyhedron $\mathcal{D}_i$. Note that this polyhedron has constraints involving only outer loop variables $x_j, j \leq i$. In the rest of this paper, we write a loop nest as $\mathcal{L}_1 :: \mathcal{L}_2 :: \cdots :: \mathcal{L}_n :: S$.

For example, consider the following polyhedral domain $\mathcal{D}$ (Fig. 5), in a program parameterized by `N` and `M`:

$\{$`i,j | j<=i<=N; j>=0; i>=M; i+j<=N+M; M>=2; M<N`$\}$

Let us decompose $\mathcal{D}$ into a loop nest, i.e., compute $\mathcal{D}_1$ and $\mathcal{D}_2$ such that

$$\mathcal{D} = \mathcal{C}_0 \ \cap \ \mathcal{D}_1 \ \cap \mathcal{D}_2$$

The context $\mathcal{C}_0$ is the polyhedron constraining the program parameters, i.e., the projection of $\mathcal{D}$ on the `N,M` dimensions:

$\mathcal{C}_0 =\{$`N,M | M>=2; M<N`$\}$

We start with the i-loop. The context for this step is $\mathcal{C}_0$. We first project out the inner dimension j, and this yields the following polyhedron:

$\{$`i,j | M<=i<=N; M>=2; M<N`$\}$

After simplification in the context of $\mathcal{C}_0$, it yields a polyhedron giving the bounds of the outer loop:

$\mathcal{L}_1 = \{$`i | M<=i<=N`$\}$

We now generate the next loop, in the context $\mathcal{C}_1 = \mathcal{C}_0 \ \cap \ \mathcal{D}_1$. This level of the recursion does not require any projection, since there are no inner loop variables. The polyhedron $\mathcal{D}$ is just simplified in the context $\mathcal{C}_1$, yielding the polyhedron:

$\mathcal{L}_2 = \{$`i,j | 0<=j<=i; j<=M+N-i`$\}$

Finally, we obtain the following loop nest:

$M \leq i \leq N$
  $0 \leq j \leq \min(i, (M + N - i))$
    loop body

4. The Multiple Polyhedra Scanning Problem

In the previous section, we discussed loop nest synthesis for a statement defined in the context of a single polyhedron. We now address the problem of generating code for multiple statements, each defined over a different polyhedron (or a finite union of polyhedra). We assume that the definition domains are independent of each other, i.e., they may, or may not, be all disjoint.

In general, data dependencies between statements imply constraints on the scanning order. Thus, we consider the problem of scanning the union of all the domains under the constraint of a given scanning order, formulated as follows:

- a *schedule* assigns a logical execution time to each index vector of each domain. We assume that this time is represented by a k-dimensional time vector[2], and is an affine function Λ of the index vector;
- the scanning order must respect the lexicographic order of the logical execution times. We define the lexicographic order $\prec$ as follows: let x and y be two n-dimensional vectors; we define

$$x \prec y \equiv \exists p,\ 1 \leq p < n,\ (x_{(1..p)} = y_{(1..p)}) \wedge (x_{p+1} < y_{p+1})$$

It follows that domains can not be scanned independently. For example, consider the domains $\mathcal{D}_1 = \{$`i| 1<=i<=10`$\}$ and $\mathcal{D}_2 = \{$`i| 4<=i<=12`$\}$, with respective schedules $\Lambda_1(i) = i+1$ and $\Lambda_2(i) = 2i$. The statements associated with points 1 and 10 of $\mathcal{D}_1$ have to be executed at logical time 2 and 11, while the statements associated with points 4 and 8 of $\mathcal{D}_2$ have to be executed at times 8 and 16. Since $1 < 8 < 10 < 16$, we cannot first completely scan $\mathcal{D}_1$, and then scan $\mathcal{D}_2$, and neither can we scan $\mathcal{D}_2$ followed by $\mathcal{D}_1$.

Hence, it is not possible to generate separate loops to scan both $\mathcal{D}_1$ and $\mathcal{D}_2$. These loops must be partially merged, so that the statements are executed in the order given by the schedule.

The simplest solution uses a perfectly nested loop to scan a convex superset (such as the bounding box) of this union. However, since the domain scanned by this loop is also a superset of *each* statement's domain, we cannot unconditionally execute the statements within the loop body. Rather, each statement S_i must be guarded by conditions which test that the current loop index vector belongs to $\mathcal{D}_i$.

The superset of the union scanned by the perfect loop may simply be the bounding box of the domain (Fig. 6.a, as done by the LooPo code generator (Griebl et al., 1998)), or the convex closure of the domain (Fig. 6.b as done by the Omega code generator (Kelly et al., 1995).

Solutions based on perfect loop nests yield compact but inefficient code because of the following limitations.

- Empty iterations: a perfectly nested loop scans a convex polyhedron. Since a domain that is a union of polyhedra may not be convex, some iterations of the loop may not execute any statements. It is often especially inefficient when a domain has an extreme aspect ratio. For example in Fig. 6.a, only about `M` out of every `N` points are usefully visited. Depending on the parameter values (say if $\mathtt{M} \ll \mathtt{N}$) and on the complexity of the loop body, this could have a significant overhead.

[2] k is the same for all domains

(a) bounding box

```
for (i=1; i<=N; ++i){
  for (j=1; j<=N; ++j){
    if (j<=M)
      S1;
    if (i==j && i>=3)
      S2;
  }
}
```

(b) convex closure

```
for (i=1; i<=N; ++i) {
  for (j=1; j<=min ((i+3N+M-4)/4,
                    i+M-1, N);
       ++j) {
    if (j<=M)
      S1;
    if (i==j && i>=3)
      S2;
  }
 }
}
```

Figure 6. Code scanning a superset of the union of domains of Fig. 1

- Control overhead: each loop iteration must test the guards of all guarded statements, causing a significant control overhead.

- Finally, finite unions of finite parameterized polyhedra do not always admit finite convex supersets. Consider for example the following parameterized domains : {`i| 1<=i<=N`} and {`i| 1<=i<=M`}; the smallest convex super-set of these domains is the infinite polyhedron {`i| 1<=i`} (note that {`i| 1<=i<=max(M,N)`} is not a convex polyhedron).

To avoid this overhead, we can separate a union of polyhedra into several distinct regions that can each be scanned using imperfectly nested loops. The resulting code is more efficient, since:

- Imperfectly nested loops can scan non-convex regions, avoiding empty iterations;

- It may be possible to choose these loops such that some statement guards become always true or always false; when a guard is always true, the guard may be removed; when a guard is always false, the entire statement can be removed. This optimization increases the ratio of the number of executed statements to the number of tested guards, and thus reduces the control overhead. This can be carried through to its logical limit to yield code with no guards.

However, the efficiency is obtained at the expense of code size for two reasons. First, a perfect loop is replaced by an imperfectly nested loop, which scans a disjoint union of polyhedra. Second, a statement domain may be divided into several disjoint polyhedra, where each polyhedron is scanned by a different loop. In this case, the statement code has to be

```
for (i = 1; i <= 2; ++i) {
  for (j = 1; j <= M; ++j) {
    S1;
  }
}
for (i = 3; i <= N; ++i) {
  for (j = 1; j <= min((-1 + i),M); ++j) {
    S1;
  }
  if (i<=M) {
    j = i;
    S1;
    S2;
  }
  if (i>=M+1) {
    j = i;
    S2;
  }
  for (j = (1 + i); j <= M; ++j) {
    S1;
  }
}
```

Figure 7. Imperfectly nested loops scanning the union of domains of Fig. 1

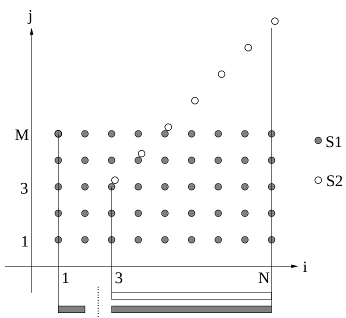
duplicated in each of these loops. Fig. 7 illustrates this for the example of Fig. 1.

5. An Algorithm to Scan Unions of Polyhedra

As described above, the simple method for scanning unions of polyhedra is to scan a convex superset using guards. The inefficiencies are then resolved by post-processing. Our solution is more holistic. It is recursive, similar to the algorithm for scanning a single polyhedron. This enables us to precisely control the tradeoff between efficiency and code size. We first present the outline of the algorithm using an illustrative example, then formally develop the algorithm, and finally describe an important subproblem, namely that of sorting polyhedra.

In this section, we assume that all loops are sequential, and defer to Sec. 6 the extension to parallel code generation and also the size-speed tradeoff. Thus the number of time dimensions k is equal to the dimension n of all polyhedra. This necessitates that all polyhedra have the same number of dimensions, but this does not introduce any loss of generality: any polyhedron of dimension n' can be extended to a polyhedron of dimension $n > n'$.

(a) Projection on `i` and separation into disjoint polyhedra



(b) Sorted `i` loops

```
{i | 1<=i<=2} ::
  {i,j | 1<=i<=2; 1<=j<=M} ::
    S1;
{i | 3<=i<=N} ::
  {i,j | 3<=i<=N; 1<=j<=M} ::
    S1;
  {i,j | 3<=i<=N;  i=j} ::
    S2;
```

Figure 8. Projections of domains on first dimension and separation into disjoint loops

5.1. Outline of the Algorithm and Illustration

Our algorithm recursively decomposes the union of polyhedra into imperfectly nested loops, starting from outermost loops to innermost loops. At each step, we seek to solve the following problem: given a context (i.e., a polyhedron in $\mathbb{Z}^{(d-1)}$ containing the outer loops and system parameters), and a union of polyhedra in $\mathbb{Z}^n, n \geq d$, generate a loop that scans this union in lexicographic order.

We generate each additional level of loops by:

1. Projecting the polyhedra onto the outermost d dimensions;

2. Separating these projections into disjoint polyhedra;

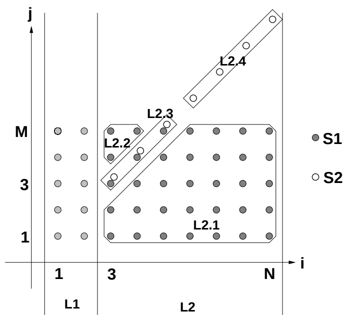
3. Recursively generating loop nests that scan each of these;

4. Sorting these loops so that their textual order respects the lexicographic order.

There are two subtle details that we will need to address. First, the idea of "sorting" polyhedra (in step 4), necessitates an order relation and an associated algorithm. Second, we need to ensure that the "separation" of polyhedra (in step 2) must be such that the resulting polyhedra can be so sorted.

Before giving the details, we illustrate the algorithm with an example. Let us start with the program of Fig. 1, and generate the first level of loops. Each domain is projected onto the first dimension and the parameters, i.e., {`i,N,M`}. These projections are then separated into three disjoint regions: one region containing only `S1`, one region

(a) Disjoint polyhedra



(b) Sorted loops

```
L1     {i | 1<=i<=2} ::
L1.1     {i,j | 1<=j<=M} ::
           S1;
L2     {i | 3<=i<=N} ::
L2.1     {i,j | 1<=j<=i-1; j<=M} ::
           S1;
L2.3     {i,j | i=j; j<=M} ::
           S1;
           S2;
L2.4     {i,j | i=j; M+1<=j} ::
           S2;
L2.2     {i,j | i+1<=j<=M} ::
           S1;
```

Figure 9. Second level: projection and separation of the second loop nest (L2)

containing both S1 and S2, and one region containing only S2. For our example, the last region is empty, as shown in Fig. 8.a. Furthermore, note that some regions (those that involve the difference of polyhedra) could be unions of polyhedra.

The two remaining regions represent two loops scanning different pieces of dimension i, parameterized by N and M. At this point, only the first level polyhedra ({i | 1<=i<=2} and {i | 3<=i<=N}) can be interpreted as loops. The other (2-dimensional) polyhedra act as guards on the statements. Note that in Fig. 8, these guards are partially redundant with their context. The redundant constraints are eliminated later.

We will recursively generate separate pieces of code to scan, respectively, regions 1 (containing S1 alone) and 2 (containing both S1 and S2). Then we will, textually place the former *before* the latter. This order respects the lexicographic schedule. Finding such a textual order is trivial for the first dimension, but becomes more complicated for subsequent levels. In the rest of this example, whenever such an order is needed, we will give a valid order without discussing how to obtain it.

Now, we generate the next level of loops in the context of the first level loops. The first i-loop (labeled L1 in Fig. 9(a)) contains only one statement, and thus the generation of its inner loop is a perfect loop generation problem. Next, consider the second i-loop, L2. It contains two guarded statements. In the next level of transformation, these domains are first separated into four disjoint polyhedra: two of them (namely L2.1 and L2.2) containing S1 alone, one (L2.3) containing both `S1` and `S2`, and one (`L2.1`) containing `S2` alone. Then, we sort these polyhedra, such that the following constraints are respected.

- Loop (`L2.1`) precedes loops (`L2.2`), (`L2.3`) and (`L2.4`).
- Loop (`L2.3`) precedes loop (`L2.2`).

There exist multiple valid textual orderings of these four loops, and we propose one of them in Fig. 9.(b). When we discuss loop sorting later in Sect. 5.3 we will describe other solutions for this example. Also note that there is no constraint on the relative position of loops (`L2.2`) and (`L2.4`), since their respective contexts ({`i|i<=M-1`} and {`i|1>=M+1`}) are distinct. In other words, for any value of the parameter `M` and the outer loop index `i`, at least one of these loops is empty; it follows that any textual order of these loops will execute correctly.

Finally, we generate code by pretty printing the sorted nested loops, yielding the code we saw previously in Fig. 7.

5.2. Formal Presentation of the Algorithm

Recall that the algorithm is recursive, starting from the outer to the inner loop indices. At each step of the recursion, given a context, i.e., a polyhedron which defines the outer loops' bounds and the parameter constraints, we generate an additional set of loops by (i) projecting the statement domains onto the context and the current loop variable, (ii) separating these projections into distinct polyhedra, (iii) recursively generating nested loops, and (iv) sorting them to determine their textual order.

The following algorithm generates one level of the loops at depth d, given a union of domain-qualified statements P_i::S_i, and a context $\mathcal{C}$.

1. Replace each polyhedron P_i by its intersection with the context $\mathcal{C}$. This ensures that P_i defines exactly the domain of statement S_i in the context of the current nested loop.

2. Compute the projection L_i of each polyhedron P_i onto the outermost d loop variables and the parameters, i.e., eliminate the inner variables. Each statement S_i is now qualified by its validity domain P_i and the projection of this domain L_i, written as L_i::P_i::S_i. Note that we do not simplify P_i in the context of L_i but retain it unchanged. This notation is thus redundant, since P_i is the domain of S_i. As explained below, the constraints of P_i will be simplified in the context at the last step of the recursion, and all remaining constraints—i.e., those that are not loop bounds—will remain as guards on S_i;

3. Separate all projections L_i into disjoint polyhedra: for example, with two loops L_i::P_i::S_i and L_j::P_j::S_j, when $L_i \cap L_j$ is non-empty, we generate three regions, containing respectively, S_1 alone, S_2 alone, and both S_1 and S_2

   – $(L_i - L_j)$::P_i::S_i
   – $(L_j - L_i)$::P_j::S_j
   – $(L_i \cap L_j)$::(P_i::S_i, P_j::S_j)

   Note that the first two regions may be unions of polyhedra (in this case, we consider each polyhedron as a region, and duplicate the associated statements). This is illustrated in Fig. 9, where the region containing `S1` alone is the union of `L2.1` and `L2.2`. We repeat the separation until the L are all distinct. Note that this step may associate several statements to a loop L.

   The separation of more than two loops is a straightforward extension and explained later.

4. Recurse for each loop $L :: (P_r :: S_r, P_s :: S_s)$, with context $\mathcal{C} \cap L$.

5. Sort the loops, as described in Sect. 5.3, to determine their textual order.

6. Finally, simplify each loop L_i in the context of $\mathcal{C}$; this step eliminates redundant loop bounds.

A couple of points merit a detailed explanation.

Deferred simplification of guards Note that we simplify loop guards only in step 6, the very end of the recursion, rather than in step 2. We illustrate how this gives us the possibility to optimize the code with the following example. Consider two statements, each one associated to a two-dimensional domain:

```
{i,j,N| 1<=i<=10; 1<=j<=i}::S1
{i,j,N| 1<=i<=10; 11<=j<=N}::S2
```

The first level of recursion generates the following program:

```
{i,N|1<=i<=10}::
  {i,j,N| 1<=i<=10;1<=j<=i}::S1
  {i,j,N| 1<=i<=10;11<=j<=N}::S2
```

Now suppose that we simplify right now the constraint on `i` in each `j`-loop, yielding

```
{i,N|1<=i<=10}::
  {i,j,N| 1<=j<=i}::S1
  {i,j,N| 11<=j<=N}::S2
```

Now, the information that $i \leq 10$ is "lost", and the recursive call is forced to to consider the case where $i \geq 11$, and thus generate three loops. However, with the knowledge of the context, it turns out that the two loops are distinct and that a third loop would be empty for any value of `i`.

Separation of multiple loops In step 3, we use the following algorithm which, given a set of polyhedra $\mathcal{P} = \{P_i\}$, computes a set of disjoint polyhedra $\mathcal{Q} = \{Q_i\}$ such that $\cup_i P_i = \cup_j Q_j$. $\mathcal{Q}$ is initially empty. Each step considers a polyhedron $P_i \in \mathcal{P}$ and computes a new $\mathcal{Q}$ as follows.

- For all $Q_j \in \mathcal{Q}$, add $Q_j - P_i$, associated with the statements of Q_j alone, to (the next round of) $\mathcal{Q}$.
- For all $Q_j \in \mathcal{Q}$, add $Q_j \cap P_i$ associated with the statements of Q_j and P_i, to (the next round of) $\mathcal{Q}$.
- Finally add $P_i - (\cup_i Q_i)$, associated to the statement of P_i alone.

Our method separates the set of statement domains into disjoint polyhedra without considering any order between these polyhedra. Indeed, the algorithm described above manipulates non ordered sets of polyhedra, and only ensures that the result is a set of disjoint polyhedra. Note that this is an essential point of our method. We show in next section that, given a set of disjoint polyhedra, we can generate and textually order loops such that these loops respect the lexicographic order.

5.3. Sorting Parameterized Disjoint Polyhedra

In this section, we address the problem of sorting disjoint polyhedral domains. The loop separation step described above does not give any information on how to order the set of loops it yields. This separation is useful only if it is always possible to find an textual order for these loops. In this section, we show that such an order always exists, and can be computed. We first define an order between polyhedra. Then we use this order to textually order loops.

The key idea of sorting two loops (or polyhedra) answers the following question: is it possible to textually place the code that scans one of them before the code that scans the other, without violating the lexicographic order? Such a textual order makes sense only if the two polyhedra share a common context, and hence this order is only a partial order. Moreover, a necessary condition is that the polyhedra are all disjoint. Using two distinct loops, it is not possible to scan in the lexicographic order two polyhedra that intersect.

A loop enclosed in a nested loop of depth l, in a program with p symbolic parameters, is specified by a 1-dimensional polyhedron parameterized by a $l+p$ vector. This vector defines the context of the loop. A correct textual order for a set of loops has to be valid for any value of the context, i.e., for any value of the program parameters and the outer loop variables.

We first define a partial order on disjoint parameterized polyhedra. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two 1-dimensional polyhedra parameterized by $(l+p)$-vectors. We note $\mathcal{C}_1$ and $\mathcal{C}_2$ the respective $(l+p)$-dimensional domains of parameters of $\mathcal{D}_1$ and $\mathcal{D}_2$. In other words, $\mathcal{C}_1$ and $\mathcal{C}_2$ are the respective projections of $\mathcal{D}_1$ and $\mathcal{D}_2$ on the parameter space. We define the order, noted $\lhd$, as follows: $\mathcal{D}_1 \lhd \mathcal{D}_2$ iff:

1. $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$
2. $\mathcal{C}_1 \cap \mathcal{C}_2 \neq \emptyset$
3. $\forall q \in \mathcal{C}_1 \cap \mathcal{C}_2, \forall x \in \mathcal{D}_1(q), y \in \mathcal{D}_2(q), x < y$,
   i.e., for any value of the context, each point of $\mathcal{D}_1$ precedes all the points of $\mathcal{D}_2$.

As stated before, the first condition implies that this partial order can be applied only to sets of disjoint polyhedra. Indeed, the condition is always ensured by the loop separation algorithm described before.

When the second condition is not verified, then $\mathcal{D}_1$ and $\mathcal{D}_2$ are not comparable. This means that any textual order of the loops which scan these polyhedra will respect the lexicographic order.

When $\mathcal{C}_1$ and $\mathcal{C}_2$ intersect, then $\mathcal{D}_1$ and $\mathcal{D}_2$ are comparable, and exactly one of the following relations is true:

- $\mathcal{D}_1 \lhd \mathcal{D}_2$
- $\mathcal{D}_2 \lhd \mathcal{D}_1$

Determining whether $\mathcal{D}_1 \lhd \mathcal{D}_2$ or $\mathcal{D}_2 \lhd \mathcal{D}_1$ can be done using basic polyhedral operations. For better understanding, we illustrate the case where $\mathcal{D}_2 \lhd \mathcal{D}_1$. We give the algorithm later. Consider the two polyhedra $\mathcal{D}_1$ and $\mathcal{D}_2$ of Fig. 10. They are distinct, thus there exists at least one hyperplane H which separates the combined context-loop space into two subspaces S_1 and S_2, containing respectively $\mathcal{D}_1$ and $\mathcal{D}_2$. Moreover, H is not parallel to the loop dimension (were it so, H would be

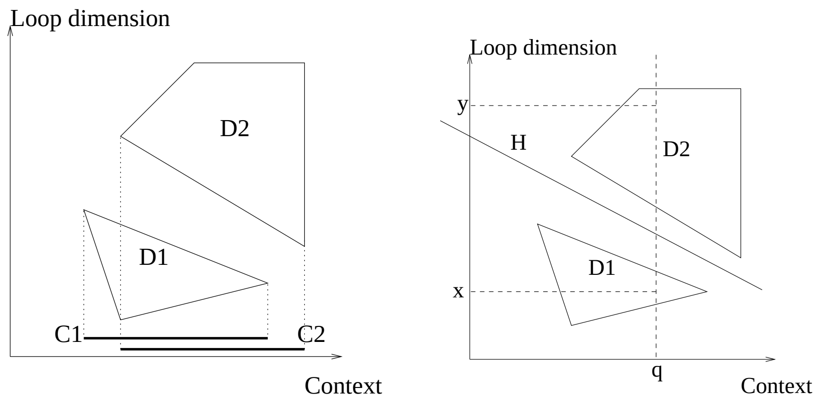


Figure 10. Precedence relation between two loops: $\mathcal{D}_1 \lhd \mathcal{D}_2$ in the context

orthogonal to the context, and thus $\mathcal{C}_1$ and $\mathcal{C}_2$ would be disjoint, a contradiction). Thus, H is such that:

$$\forall q \in \mathbb{Z}^{(l+p)}, \forall x, y \in \mathbb{Z} \text{ s.t. } \begin{pmatrix} q \\ x \end{pmatrix} \in S_1 \text{ and } \begin{pmatrix} q \\ y \end{pmatrix} \in S_2, \text{ then } x < y$$

In particular, this also holds when $\begin{pmatrix} q \\ x \end{pmatrix} \in \mathcal{D}_1$ and $\begin{pmatrix} q \\ y \end{pmatrix} \in \mathcal{D}_2$

The following algorithm computes the precedence relation between two parameterized polyhedra $\mathcal{D}_1$ and $\mathcal{D}_2$. A precondition is that these polyhedra do not intersect. The result of the algorithm is either "incomparable", $\mathcal{D}_1 \lhd \mathcal{D}_2$ or $\mathcal{D}_2 \lhd \mathcal{D}_1$.

1. Compute the projections $\mathcal{C}_1$ and $\mathcal{C}_2$ of $\mathcal{D}_1$ and $\mathcal{D}_2$ on the context space.

2. Intersect the projections: $\mathcal{I} = \mathcal{C}_1 \cap \mathcal{C}_2$.

3. If the intersection $\mathcal{I}$ is empty, then the polyhedra are not comparable, exit.

4. The polyhedra are comparable, and either $\mathcal{D}_1 \lhd \mathcal{D}_2$, or $\mathcal{D}_2 \lhd \mathcal{D}_1$.

5. Arbitrary assume that $\mathcal{D}_1 \lhd \mathcal{D}_2$. By definition, $\mathcal{D}_1 \lhd \mathcal{D}_2$ implies that

$$\forall p \in \mathcal{I}, \forall x_1 \in \mathcal{D}_1(p), \forall x_2 \in \mathcal{D}_2(p), \exists x \text{ such that } x_1 < x < x_2$$

   Let $\mathcal{H}(p) = \{x \mid x_1 < x < x_2, x_1 \in \mathcal{D}_1(p), x_2 \in \mathcal{D}_2(p)\}$. To verify that, for all p, this polyhedron is not empty, it is sufficient to verify that the polyhedon $\mathcal{P}$, defined as following, is not empty:

   – Select from $\mathcal{D}_1$ the constraints "$x > \ldots$"

- Select from $\mathcal{D}_2$ the constraints "$x < \ldots$"
- Compute the resulting polyhedron $\mathcal{P}$.

$\mathcal{P}$ includes $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{H}$. If $\mathcal{P}$ is empty, then the assumption was false, i.e., $\mathcal{D}_2 \lhd \mathcal{D}_1$, else it is verified, and thus $\mathcal{D}_1 \lhd \mathcal{D}_2$.

This algorithm is implemented in our polyhedral library. It requires three polyhedral operations (i.e., six calls to the Chernikova algorithm) to decide whether the two polyhedra are comparable, and one more operation to compute the order.

Given this partial order, a method to choose a textual order (out of possibly many solutions) computes a precedence graph of the set of polyhedra, and select a traversal of this graph. To build this graph, whose nodes are the polyhedra, one has to compare every pair of polyhedra, and set an edge between the corresponding nodes if the polyhedra are comparable. For x polyhedra, building this graph requires $x(x-1)/2$ comparisons. However, this is only an upper bound on the effective number of comparisons, since some comparisons can be avoided by applying transitivity. Note that the worst case is when no pair of polyhedra can be compared—the resulting graph has no edges, thus any textual order is valid, but deciding this requires the maximum number of comparisons.

Finally, the following algorithm selects a valid order:

1. Let $i = 0$;
2. Find a polyhedron which has no predecessor, mark it with i and remove it from the graph;
3. Increment i and repeat step 2 until there is no polyhedron left.
4. Each polyhedron is now labeled with a position. The generated loops can be textually placed following the ascending order of these positions.

We illustrate this on our running example. We describe the computation of the set of solutions for sorting the loops in the second step of the recursion (Fig. 9.(a)). Loop `L2` contains four distinct j-loops, noted `L2.1`, `L2.2`, `L2.3` and `L2.4` (Fig. 11).

- For any value of `i` and of the parameters, loop `L2.1` precedes all the other loops, i.e., its corresponding node is a root of the precedence graph.
- Now ignore loop `L2.1`. Loops `L2.3` and `L2.4` have no predecessors, and thus can be placed immediately after `L2.1`. At this time, we already have two partial solutions:

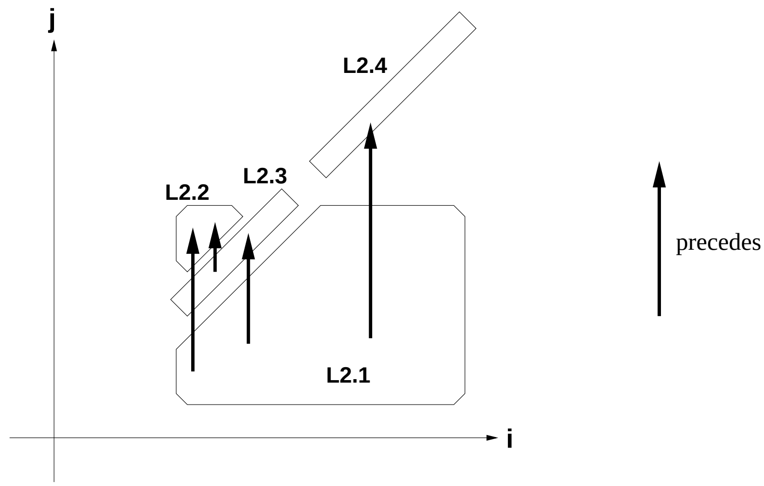


Figure 11. Partial order between parameterized polyhedra of Fig. 9.(a)

- `L2.1`; `L2.3` or
- `L2.1`; `L2.4`.

– Let us consider the first solution. The two remaining loops `L2.2` and `L2.4` are not comparable, so their textual order is meaningless. In other words,

  - `L2.1`; `L2.3`; `L2.2`; `L2.4`, and
  - `L2.1`; `L2.3`; `L2.4`; `L2.2`

  are both valid textual orders.

– Let us now consider the other solution (`L2.1`; `L2.4`). The remaining loops `L2.2` and `L2.3` admit an order, and thus

  - `L2.1`; `L2.4`; `L2.3`; `L2.2`

  is the third valid order.

6. Extensions

We now address a number of extensions to the main algorithm. These extensions are implemented in our loop generator.

The algorithm described in Sect. 5 generates imperfectly nested loops, optimized for efficiency. It removes all the guards on the s-tatements and, consequently, ensures that each iteration of a loop will

```
L1     {i | 1<=i<=2} ::
L1.1     {i,j | 1<=j<=M} ::
           S1;
L2     {i | 3<=i<=M} ::
L2.1     {i,j | 1<=j<=i-1; j<=M} ::
           S1;
L2.3     {i,j | i=j; j<=M} ::
           S1;
           S2;
L2.2     {i,j | i+1<=j<=M} ::
           S1;
L3     {i | i=M+1} ::
L3.1     {i,j | 1<=j<=i-1; j<=M} ::
           S1;
L3.3     {i,j | i=j; j<=M} ::
           S1;
           S2;
L4     {i | 3<=i<=M} ::
L4.1     {i,j | 1<=j<=i-1; j<=M} ::
           S1;
L4.4     {i,j | i=j; M+1<=j} ::
           S2;
```

Figure 12. Second separation of loop `L2` into `L2`, `L3` and `L4`

execute all the statements it contains. However, it does not ensure that all loops are visited in a given context. To illustrate this limitation, consider our running example in Fig 9. Loop `L2.2` is empty in the context $\mathtt{i}\geq\mathtt{M}$, and loop `L2.3` is also empty when $\mathtt{i}\geq\mathtt{M+1}$. In other words, the code generated for loops `L2.2` and `L2.3` is dead code for some of the iterations of `L2`. If $N \ll M$, these two empty loops represent a significant execution overhead.

This is known at compile time, and thus the generated code can be further optimized. This problem is resolved by adding a second separation procedure in the bottom-up phase of the recursion, between steps 5 and 6 of the algorithm. By projecting every loop on the context space, we obtain for each loop, the sub-context where it is not empty. Then we separate these sub-contexts, using the same separation algorithm as for the loops. Finally, we sort the separated contexts. For the loop program of Fig 9, this extended algorithm generates the loops given in Fig. 12. Note that the overhead is removed at the expense of code size.

6.1. Trade-off between code size and efficiency

At the other extreme, another extension of our algorithm can limit code size. Recall that our method is recursive, and that at each step of the recursion we separate the polyhedra into disjoint ones in order to generate imperfectly nested loops. We can always choose to not separate the polyhedra and simply generate loops with guards. We implemented a simple heuristic which considers that the depth of a guard indicates its cost. A similar heuristic is also implemented in the Omega code generator (Kelly et al., 1995). The heuristic kicks in for all loops whose depth is smaller than the chosen one. This corresponds to generating perfectly nested loops which scan the outermost dimensions, and then generating optimized loops for the inner dimensions.

However, the estimate is not realistic. The cost of a guard is the number of times it is potentially evaluated, i.e., the volume of its context polyhedron. Since there are well known techniques for estimating the number of integer points in a parameterized polyhedron using Ehrhart polynomials (Clauss and Loechner, 1996), we could use an exact function rather then a heuristic. This is the subject of our ongoing work.

6.2. Non unimodular mappings

The polyhedral model can also deal with non unimodular mappings. We use a generalization of the change of basis transformation to apply non unimodular mappings to polyhedra. The intuition behind this is that any non-singular matrix M admits a unimodular row extension U such that U admits a unimodular column extension. U specifies a one-to-one transformation. If we are given a non unimodular mapping M, we apply a change of basis using U. The image of a n-dimensional polyhedron P by U is a m-dimensional polyhedron Q, where $m \geq n$. The change of basis by U ensures that P and Q have the same number of integer points, and moreover that the n first dimensions of P correspond to the mapping specified by M.

For example, consider the polyhedral domain {`i|1<=i<=N`}, where `N` is a program parameter, and the transformation `(i->2i+1)`. This transformation is not unimodular, but the extended transformation `(i->2i+1,i)` is one-to-one (its inverse is `(i,j->j)`). The image of the domain by this extended transformation is

{`i,j | 3<=i<=2N+1; i=2j+1`}

This polyhedron is embedded in two-dimensional space, but it is also constrained by an equality, and thus defines a one-dimensional set of points. In other words, it can be scanned using a single loop, and the stride of this loop can be determined easily, using polyhedral operations.

6.3. Data-parallel code

We now address the generation of code where the outermost k loops are sequential, and the other loops are parallel.

We introduce a method to generate data-parallel code as a generalization of our method of sequential code generation. We extend the algorithm described in previous section to the case where the schedule dimension k is lower that (or equal to) the dimensions of definitions domains.

Given a k-dimensional schedule Λ_V for a variable V, we compute a row extension T_V of Λ_V such that T_V admits a unimodular column extension. Hence T_V defines a one-to-one transformation, and can be used to change the basis of V. Since the k first rows of T_V are the rows of Λ_V, the first k dimensions of V in the new basis now carry all the data dependencies. In other words, only the first k depths of loops need to respect the lexicographic order of the k first indices. We generate imperfectly nested loops of depth k, using the algorithm described in Sect. 5. Then, since there are no more data dependencies, we generate perfectly nested loops which scan the remaining dimensions of the definition domain of each statement, and we annotate these loops such that they can be recognized as parallel loops by the back-end code generator.

7. Related Work

The problem of loop synthesis was first addressed by Ancourt and Irigoin (Ancourt and Irigoin, 1991). They use Fourier-Motzkin pairwise elimination to produce bounds at each level. The idea is to use the Fourier-Motzkin pairwise elimination algorithm to find the bounds for each loop. The system of constraints is transformed into a triangular system by projecting out, for each loop, the inner loop indices. This algorithm yields a system where each loop index depends only on the outer loop indices and the program parameters. The main drawback of this method is that it creates redundant bound equations, that have to be eliminated afterwards.

Fourier-Motzkin based methods are used by a number of parallelizing compilers, including the Pandore II compiler (Andre et al., 1995), SUIF (Amarasinghe, 1997) and the PIPS project (Irigoin et al., 1991).

Li and Pingali (Li and Pingali, 1994) extend the previous method to deal with non unit loop strides, by allowing non unimodular transformations. Their method is based on the Hermite Normal Form of matrices (Schrijver, 1986): they decompose the transformation into two parts, a unimodular one, and non-unimodular which preserves the lexicographic order. They generate a nested loop scanning the image of the source index domain by the unimodular part of the transformation. Then, in the loop body, the variables of the loop are replaced by their image by the other part of the transformation. Ramanujam (Ramanujam, 1992; Ramanujam, 1995) and Xue (Xue, 1994) also presented methods to deal with non unit loop strides. They use the Fourier-Motzkin elimination to compute the loop bounds, and the Hermite decomposition to determine the strides. All these methods generate loops that scan exactly the image of the original domain by the non-unimodular transformation, i.e., a Z-polyhedron. Moreover, they are restricted to scanning a single domain. In contrast our method, which deals with multiple polyhedra, uses the Smith Normal Form to compute polyhedra whose projections are the respective Z-polyhedra, and generates a loop nest which scans these polyhedra.

Collard, Risset and Feautrier (Collard et al., 1995) show how the dual simplex method (Schrijver, 1986; Feautrier, 1988) can be used to generate loop bounds. It has been implemented in Bouclettes, a Fortran parallelizer (Boulet, 1996).

All the above methods deal essentially with the problem of scanning a single polyhedron. There has not been much work on the multiple polyhedra scanning problem. To the best of our knowledge, only four other authors have addressed this problem.

Our initial ideas on polyhedra separation and the intuition behind sorting were briefly presented by Quinton et al. (1995), and implemented in an early prototype ALPHA compiler. However, the sorting step was not completely formalized, and the generator did not correctly handle domains with equalities. A similar method for loop separation was independently developed by Albiez (Albiez, 1998).

Collard (Collard, 1994) addresses the generation of loops from multiple polyhedra, with emphasis on the case of loops with incompatible strides. The generated code contains many guards. This paper assumes that there is a single time dimension, but the schedule may not admit a unimodular completion. The author does not therefore face the problem of polyhedron separation and/or sorting, but must confront non-unimodular transformations. Much of the paper is devoted to the (non-unit) strides due to the schedule, and to the removal of redundant guards.

The LOOPO project (Griebl and Lengauer, 1996) at the University of Passau also explores the parallelization in the polyhedral model and its extension and addresses the problem of code generation. The currently available working generator produces simplistic code that scans the bounding box of the union of all the domains and executes guarded statements. In a recent paper, they also address the problem of imperfectly nested loop generation (Griebl et al., 1998). Given a set of domain-qualified statements, they seek to recursively merge the polyhedra, in much the same way as we do. It seems however, that they do not exploit the *complete* context information that is available which leads to naive code. They claim that, with unknown program parameters, the number of branches—loop sequences—in the final program is an exponential of the number of unknown loop bounds (exact number of permutation). Their preliminary experimentation were discouraging, as evidenced by their statement that, "a simple program of a few lines with three nested loops [... leads to code with an] `if`-cascade of 40320 branches". We have seen in this paper that generating code for every permutation is not necessary, since parameterized polyhedra admit a partial order.

The Omega system (Kelly and Pugh, 93; Kelly et al., 1995), which is also freely available under the Gnu public license, produces code to scan unions of convex sets, in an extension of the polyhedral model. Although the generated code is very similar to ours, it uses a different, non recursive method. It starts from the naive code (a perfect nest of loops which scan the convex closure of the union of domains), and then iteratively tries to remove sources of overhead (guards and min/max operations in loop bounds). However, it generates incorrect code for domains with equalities, and because the generator is no longer maintained, it is difficult to determine whether this is a simple bug or an inherent limitation. The most interesting aspect of their code generator is their technique of post-optimization. As described before, they seek using a bottom-up method to "lift" guards outside loops. Essentially, an affine guard is expressed as an inequality between the current loop index and an affine function f of all the surrounding ones. We can split the loop into two by breaking it at f and, in each of the two parts, we can be sure that the guard is either always true or always false. Hence we can unconditionally suppress the statement in one of the parts and unconditionally execute it in the other. Since the tests are done one guard at a time and one dimension at a time, they experience considerable code explosion. The trade-off between code size and code efficiency is controlled by specifying the depths from which overhead is eliminated.

Finally, let us mention another optimization, which is independent of those presented here. Since loop bounds are minimum or maximum of affine functions of outer loop variables and parameters, the bounds testing may also lead to overheads. Chamski (Chamski, 1993) describes a method to eliminate minimum and maximum expressions, by statically computing these expressions and generating imperfect sequences of loops. This is implemented in our code generator as a post-optimization, since it is completely orthogonal to the methods presented here.

8. Conclusion

We have addressed the problem of scanning the union of (not necessarily disjoint) polyhedra in lexicographic order their indices. The problem arises in the context of the code-generation phase of parallelizing compilers, especially when one permits different mappings to be applied to the individual statements in the body of a static control loop. Our specific context is that of compiling ALPHA, a data-parallel functional language based on systems of affine recurrence equations defined over polyhedral domains. The MMALPHA system for manipulating ALPHA programs (written using the Mathematica system) is available under the Gnu public license (see `http://www.irisa.fr/cosi/ALPHA`). The loop generation module in the ALPHA code generator is written in C++ and is also available under the Gnu public license (at `http://www.irisa.fr/cosi/ALPHA/codegen`).

Our solution is based on a recursive algorithm to generate imperfectly nested loops. The key idea is that at each level of the recursion (responsible for generating a set loops at that level), we use two main operations, namely *separation* of the polyhedra, and *sorting* polyhedra in a given context. Because of a modular separation of the two operations, and because they are applied only in a limited recursive context, we avoid the explosion in the number of comparisons that other methods seem to perform. Furthermore, our algorithm allows fine control of the tradeoff between code size and code efficiency.

We use a unified representation for loops and guards, which ensures that bounds and remaining guards are not redundant, and our algorithm is based on a small set of polyhedral operations drawn from Polylib, a library of polyhedral operations (Wilde, 1993).

In the context of the MMALPHA system, our ongoing work involves the code generation problem from hierarchical ALPHA programs, which will involve generation procedure calls and memory allocation issues. In addition, in the context of silicon compilation for dedicated VLSI implementations we are also exploring the problem of automatic generation of interfaces. We are also investigating extensions to the code generation procedure that will allow code to be generated under precise code size constraints.

We have proposed a solution to the problem of sorting parameterized loops, and shown that any set of disjoint parameterized loops admit a textual order that respects a given lexicographic schedule. We have defined a partial order on parameterized polyhedra, and used this order to sort loops.

Acknowledgment

The authors thank the anonymous reviewers for their useful comments on an earlier draft of this paper.

References

Albiez, O.: 1998, 'Parcours d'une superposition de domaines'. In: *RenPar'10*.

Amarasinghe, S. P.: 1997, 'Parallelizing Compiler Techniques Based on Linear Inequalities'. Ph.D. thesis, Stanford University.

Ancourt, C. and F. Irigoin: 1991, 'Scanning Polyhedra with DO Loops'. In: *Third Symposium on Principles and Practice of Parallel Programming (PPoPP)*. pp. 39–50.

Andre, F., M. Le Fur, Y. Maheo, and J.-L. Pazat: 1995, 'The Pandore data-parallel compiler and its portable runtime'. *Lecture Notes in Computer Science* **919**, 176–183.

Banerjee, U.: 1993, *Loop Transformations for Restructuring Compilers: The Foundations*. Norwell, Mass.: Kluwer Academic Publishers.

Boulet, P.: 1996, 'Bouclettes: A Fortran Loop Parallelizer'. *Lecture Notes in Computer Science* **1067**, 784–791.

Chamski, Z.: 1993, 'Nested loop sequences: towards efficient loop structures in automatic parallelization'. Technical Report PI 772, Irisa France.

Chernikova, N.: 1965, 'Algorithm for finding a general formula for the non-negative solution of a system of linear inequalities'. *U.S.S.R Computational Mathematics and Mathematical Physics* **5**(2), 228–233.

Clauss, P. and V. Loechner: 1996, 'Parametric Analysis of Polyhedral Iteration Spaces'. In: *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*.

Collard, J.-F.: 1994, 'Code Generation in Automatic Parallelizers'. In: C. Girault (ed.): *Proc. of the Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G 10.3*. Caracas, Venezuela, pp. 185–194.

Collard, J.-F., T. Risset, and P. Feautrier: 1995, 'Construction of DO loops from systems of affine constraints'. *Parallel Processing Letters* **5**(3), 421–436.

Darte, A. and Y. Robert: 1993, 'Mapping uniform loop nests onto distributed memory architectures'. rapport de recherche 93-03, LIP.

Darte, A. and F. Vivien: 1996, 'Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs'. rapport de recherche 96-06, LIP, ENS Lyon.

Feautrier, P.: 1988, 'Parametric Integer Programming'. *Rairo Recherche Opérationnelle* **22**(3), 243–268.

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

Feautrier, P.: 1992, 'Some efficient solutions to the affine scheduling problem, part I, one dimensional time, and part II, multidimensional time'. *International Journal of Parallel Programming* **21**(5-6), 313–348, 389–420.

Feautrier, P.: 1994, 'Toward automatic distribution'. *Parallel Processing Letters* **4**(3), 233–244.

Griebl, M. and C. Lengauer: 1996, 'The Loop Parallelizer LooPo'. In: M. Gerndt (ed.): *Proc. Sixth Workshop on Compilers for Parallel Computers*, Vol. 21 of *Konferenzen des Forschungszentrums Jülich*. Forschungszentrum Jülich, pp. 311–320.

Griebl, M., C. Lengauer, and S. Wetzel: 1998, 'Code Generation in the Polytope Model'. In: *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*. pp. 106–111.

Irigoin, F.: 1988, 'Code generation for the hyperplane method and for loop interchange'. Technical Report ENSMP-CAI-88-E102/CAI/I, Ecole Nationale Superieure des Mines de Paris.

Irigoin, F., P. Jouvelot, and R. Triolet: 1991, 'Semantical Interprocedural Parallelization: An Overview of the PIPS Project'. In: *Conference Proceedings, 1991 International Conference on Supercomputing*. Cologne, Germany, pp. 244–251.

Karp, R., R. Miller, and S. Winograd: 1967, 'The Organization of Computations for Uniform Recurrence Equations'. *Journal of the Association for Computing Machinery* **14**(3), 563–590.

Kelly, W. and W. Pugh: 93, 'A Framework for Unifying Reordering Transformations'. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland.

Kelly, W., W. Pugh, and E. Rosser: 1995, 'Code Generation for Multiple Mappings'. In: *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*. McLean, VA.

Kuck, D. J.: 1978, *The Structure of Computers and Computations.* NY: J. Wiley and Sons.

Le Verge, H., V. Van Dongen, and D. K. Wilde: 1994, 'Loop nest synthesis using the polyhedral library'. Technical Report PI 830, Irisa, France.

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

Motzkin, T. S., H. Raiffa, G. L. Thompson, and R. M. Thrall: 1953, *Contribution to the theory of games*, Chapt. The Double Description Method. Princeton University Press.

Quinton, P., S. Rajopadhye, and D. Wilde: 1995, 'Deriving Imperative Code from Functional Programs'. In: *7th Conference on Functional Programming Languages and Computer Architecture*. La Jolla, CA, pp. 36–44.

Quinton, P. and V. Van Dongen: 1989, 'The Mapping of Linear Recurrence Equations on Regular Arrays'. *Journal of VLSI Signal Processing* **1**(2), 95–113.

Rajopadhye, S. and R. Fujimoto: 1990, 'Synthesizing systolic arrays from recurrence equations'. *Parallel computing* **14**, 163–189.

Ramanujam, J.: 1992, 'Nonunimodular Transformations of Nested Loops'. In: *Supercomputing '92*. Minneapolis, Minn., pp. 214–223.

Ramanujam, J.: 1995, 'Beyond Unimodular Transformation'. *The Journal of Supercomputing* **9**(4), 365–389.

Schrijver, A.: 1986, *Theory of Linear and Integer Programming.*, A Wiley-Interscience publication. Chichester, New York: Wiley.

Wilde, D.: 1993, 'A Library for Doing Polyhedral Operations'. Technical Report PI 785, Irisa, France.

Wolf, M. and M. Lam: 1991, 'Loop transformation theory and an algorithm to maximize parallelism'. *IEEE Transactions on Parallel and Distributed Systems* **2**(4), 452–471.

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