Code Generation in the Polytope Model

Martin Griebl, Christian Lengauer, Sabine Wetzel
Fakultät für Mathematik und Informatik, Universität Passau
D-94030 Passau, Germany
{griebl,lengauer,wetzel}@fmi.uni-passau.de

Abstract

Automatic parallelization of nested loops, based on a mathematical model, the polytope model, *has been improved significantly over the last decade: state-of-the-art methods allow flexible distributions of computations in space and time, which lead to high-quality parallelism. However, these methods have not found their way into practical parallelizing compilers due to the lack of code generation schemes which are able to deal with the new-found flexibility. To close this gap is the purpose of this paper.*

1. Introduction

In recent years, methods for automatic parallelization of nested loops based on a mathematical model, the *polytope model* [9, 12], have been improved significantly. The focus has been on identifying good schedules, i.e., distributions of computations in time, e.g., [6, 8], and allocations, i.e., distributions of computations in space, e.g., [5, 14]. Thus, the *space-time mapping*, i.e., the combination of schedule and allocation, derived by state-of-the-art techniques often describes a very efficient parallel execution of the source loop nest.

In contrast, code generation has been neglected and has fallen behind in flexibility compared to modern schedulers and allocators. Therefore, many space-time mapped source loop nests cannot be expressed as a target loop nest with standard code generation techniques [1, 3, 11]. The goal of this paper is to present a code generation scheme which fills this gap between space-time mappings and code generation.

The most important recent extensions to the polytope model which code generation algorithm must catch up with are:

- the *piecewise affinity* of functions such as schedules and allocations,
- *imperfectly nested* loops, i.e., loop nests in which not all statements belong to the innermost loop and,
- the space-time mapping of *individual statements* as opposed to the whole loop body.

In addition, schedule and allocation are usually computed independently. Therefore, the *space-time matrix*, which is composed of the rows for the schedule and the allocation, does not satisfy constraints such as unimodularity (determinant of ± 1), or even invertibility. Thus, code generation must be able to deal with non-unimodular and even with singular matrices. We are not aware of the existence of such a generally applicable method.

[11] can deal with statementwise, even non-unimodular transformations, but they must be non-singular; our methods, to be presented here, could be easily integrated as a preprocessing phase into the code generation algorithm of [11]. The method of [4] can deal with by-statement mappings but none of the other extensions.

Our paper summarizes the main results of the diploma thesis by Sabine Wetzel [17]. For more examples and a more technical description we refer to her thesis. The paper is organized as follows. Section 2 introduces some important definitions and describes the notion of a *program part*, which is the basis of our code generation scheme. Section 3 presents the code generation for separate program parts, thereby solving the problems incurred by the possibly incomplete rank of the space-time matrix. Section 4 proposes two different ways of merging the separate program parts: a run-time solution (which causes a lot of control overhead at run time) and a compile-time solution. Section 5 concludes the paper with a comparison of the two presented merge algorithms.

2. Definitions and Notation

Our mathematical notation follows Dijkstra [7]. Quantification over a dummy variable x is written $(Q\,x\ :\ R.x\ :\ P.x)$. Q is the quantifier, R is a predicate in x representing the range, and P is a term that depends on x.

Let S be a statement, possibly in the body of a loop nest. If S is surrounded by loops, then multiple instances of the single statement S are executed. We call these instances *operations* and denote them with $\langle S, i\rangle$, where i is the index vector. We denote the set of all operations with Ω. The dependences between different operations are given by a dependence graph (Ω, E). The set of all index vectors for statement S is called the *index space* and denoted by $\mathcal{I}_S$.

Definition 2.1 (Schedule, allocation, space-time matrix)
Let Ω be a set of operations, (Ω, E) their dependence graph, and r, r' integer values.

- Function $t : \Omega \to \mathbb{Z}^r$ is called a *schedule* if it preserves the data dependences:

$$(\forall\, x, x' : x, x' \in \Omega \wedge (x, x') \in E : t(x) <_{\text{lex}} t(x'))$$

- Any function $a : \Omega \to \mathbb{Z}^{r'}$ can be interpreted as an *allocation*.

We require the additional restriction that schedule and allocation are piecewise affine functions for every statement S in its index space $\mathcal{I}_S$. Therefore, for every subdomain $\mathcal{S}_S \subseteq \mathcal{I}_S$, we define:

$$(\exists\, \lambda_S, \alpha_S : \lambda_S \in \mathbb{Z}^{r \times d} \;\wedge \alpha_S \in \mathbb{Z}^r \;: (\forall\, i : i \in \mathcal{S}_S : t(\langle S, i\rangle) \;=\; \lambda_S\, i + \alpha_S))$$

$$(\exists\, \sigma_S, \beta_S : \sigma_S \in \mathbb{Z}^{r' \times d} \wedge \beta_S \in \mathbb{Z}^{r'} \;: (\forall\, i : i \in \mathcal{S}_S : a(\langle S, i\rangle) \;=\; \sigma_S\, i + \beta_S))$$

The matrix $\mathcal{T}_S$ formed by λ_S and σ_S is called a *space-time matrix*:

$$\mathcal{T}_S = \begin{pmatrix} \lambda_S \\ \sigma_S \end{pmatrix}$$

For every statement S and everyone of its subdomains $\mathcal{S}_S$, the set $\mathcal{T}_S\, \mathcal{S}_S$ is called a *target polytope*; we call the corresponding loop nest which enumerates this target polytope a *target program part*.

With these definitions, we can now explain how to derive target program parts for target polytopes.

3. Generating Target Program Parts

We proceed in two steps. Section 3.1 presents briefly the basic idea and the state of the art of code generation methods. Section 3.2 presents our new extension which can deal with arbitrary space-time matrices.

3.1. State of the art

Seminal work on target loop generation considered only unimodular transformation matrices [1]. The basic idea is to substitute the source loop indices in the description of the index set by the transformed target indices. Technically, this is a *change of basis* [1]. More formally, if the source index space w.r.t. to source loop indices x is given by the source polytope $A\,x \le b$, then the target polytope w.r.t. target indices $y = \mathcal{T}\,x$ is given by $(A\,\mathcal{T}^{-1})\,y \le b$ [12].

After this change of basis, Fourier-Motzkin elimination [2, 16] can be used to generate the loop bounds; this technique eliminates variables from a system of inequalities, which is necessary since the bounds of outer loops must not depend on the indices of inner loops.

So far, this method has been extended to deal with non-unimodular invertible matrices [13, 19]. In this case, the inequality system $(A\,\mathcal{T}^{-1})\,y \le b$ enumerates the convex hull, i.e., a proper superset of the images of the source index space under $\mathcal{T}$. It is possible to pick the correct index points by using non-unit strides for the target loops. A more detailed technical description of the method, including further extensions—e.g., how to deal with non-integer coefficients in the schedule—can be found in Wetzel's thesis [17].

On the other hand, the topic of relaxing the invertibility requirement has not been addressed so far, probably because it is indispensable for deriving the source indices from the target indices in the loop bodies as well as for computing the (convex hull of the) target polytope. However, this case appears in practice, due to the unrelated computation of schedule and allocation.

```
         for $i := 0$ to $n$
            for $j := 0$ to $i-1$
$L$ :          $l[i,j] = f(l[i,j-1])$
            endfor
            for $k := 0$ to $i-1$
$U$ :          $u[i,k] = g(u[i-1,k], l[i,k])$
            endfor
         endfor
```

Figure 1. A source program leading to a singular space-time mapping

Example 3.1
Consider the source program in Figure 1.

Statement L has uniform dependences along dimension j, whereas the dependences of statement U are uniform along dimension i; furthermore, there is a dependence from $\langle L, (i,j)\rangle$ to $\langle U, (i,k)\rangle$, for $j = k$. The optimal schedules are $t(\langle L, (i,j)\rangle) = j$ and $t(\langle U, (i,k)\rangle) = i$. Any non-trivial allocation (i.e., using more than one processor) can eliminate at most two of the three dependences; the least number of remaining communications is achieved by the allocation $a(\langle L, (i,j)\rangle) = j$ and $a(\langle U, (i,k)\rangle) = k$. The resulting space-time matrices are $\mathcal{T}_L = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$ and $\mathcal{T}_U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Because of the singularity (more precisely: the non-injectivity) of $\mathcal{T}_L$, some processors will have to perform several operations at the same time step.

The next section explains how to deal with this situation.

3.2. Arbitrary space-time matrices

Our solution consists of the following steps:

1. Eliminate linearly dependent rows in the matrix (Section 3.2.1).
2. Extend the matrix to a square invertible matrix, i.e., to a basis (Section 3.2.2).
3. Generate code w.r.t. the resulting modified matrix (Sections 3.2.2 and 3.1).
4. Insert code for the rows eliminated in step 1 (Section 3.2.3).

For simplicity, we assume that the rows of the space-time matrix are ordered from top to bottom according to the desired outside-in order of the target loops. (Note that this is just a convention, not a restriction.)

3.2.1 Eliminating linearly dependent rows

In a first step, we eliminate, iteratively from top to bottom, all linearly dependent rows and store them together with their row number for further processing later on (Section 3.2.3). Technically, we use Echelon reduction [2] to determine whether row μ is linearly dependent on rows $1, \cdots, \mu-1$. The result of this elimination process is a matrix $\mathcal{T}'$ which has full row rank but is not necessarily square: the number of rows is less or equal to the number of columns.

3.2.2 Extending to a square matrix with full rank

Matrix $\mathcal{T}'$ can have fewer rows, i.e., target dimensions, than there are columns, i.e., source dimensions. To obtain an invertible square matrix, we extend $\mathcal{T}'$ to a basis by simply adding linearly independent unit vectors at the bottom of $\mathcal{T}'$, each of which spawns one missing dimension. The result is an invertible square matrix $\widetilde{\mathcal{T}}$. This matrix can be used to derive the target program parts from the target polytopes by standard methods (including the extensions for non-unimodular transformations), as described in Section 3.1.

Interpretation Before continuing with our description of the code generation method, let us briefly reflect on the changes from $\mathcal{T}$' to $\widetilde{\mathcal{T}}$. How can we interpret the artificial rows which we have added and which correspond to target dimensions but are given neither by the schedule nor by the allocation?

Since these dimensions are not laid out in time, they obviously will not carry a dependence (otherwise, the schedule would be incorrect). Therefore, we could lay them out in space.

On the other hand, the allocator did not distribute iterations along these dimensions. Therefore, in order to preserve the effect of the allocation, we decide to lay these artificial dimensions out in time and put the respective loops inside the loops enumerating the schedule. I.e., we refine the time given by the scheduler with additional dimensions. This changes neither the global schedule (which is respected by the outermost, i.e., dominant loops on time), nor the allocation (since it is not modified).

The remaining question is: what happens with the rows which have been eliminated? The next subsection will rectify the fact that we have ignored those dimensions.

3.2.3 Reinserting eliminated dimensions

First, note that a target dimension given by row μ, which is linearly dependent on rows $1, \cdots, \mu-1$ of $\mathcal{T}$, collapses to a singleton, the value r_μ of which can be computed from the coordinates $r_1, \cdots, r_{\mu-1}$ in dimensions $1, \cdots, \mu-1$. This leads directly to our solution: we compute r_μ from $r_1, \cdots, r_{\mu-1}$ and insert a loop enumerating only the singleton r_μ. The nesting level at which this loop is inserted is given by the previously stored number of the eliminated row (Section 3.2.1). The type of the loop is sequential if the eliminated row was produced by the scheduler, and parallel if it is a part of the allocation.

This way, the outermost target loops enumerate precisely all coordinates according to the given space-time mapping, i.e., the loop nest implements $\mathcal{T}$ correctly, whereas the innermost, artificially added target loops spawn the missing dimensions, which is, in general, necessary to be able to enumerate all transformed source index coordinates.

In Example 3.1, $\mathcal{T}'_L = (0 \;\; 1)$ and $\widetilde{\mathcal{T}}_L = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Intuitively, time has been refined to enable an allotment of only one computation per processor and time step.

4. Merging Target Program Parts

At this point, we have several target polytopes $P_1, \ldots, P_k$, one per statement and per subdomain (in the case of piecewise affine schedules or allocations), and their according target program parts w.r.t. invertible space-time matrices.

The complete target program must enumerate the union of these polytopes: the *quasi-convex polytope* $P = P_1 \cup \ldots \cup P_k$ [20]. The goal of this section is to derive this target program $\mathcal{P}'$. We present two different methods to this effect:

- The most general possibility is the run-time solution described in Section 4.1.
- A less costly possibility (at a first sight!) is the compile-time solution described in Section 4.2.

Since there is no loop which enumerates both space and time, we can only merge loops of the same type (sequential or parallel). This restriction is necessary for the run-time solution as well as for the compile-time solution. However, for two polytopes to be merged, the sequence of types of loops for one polytope is allowed to be a prefix of the sequence of types of loops for the other polytope. Note that the type of loops does not influence the merge technique.

4.1. Run-time solution

The run-time solution can briefly be described as follows: we construct the convex hull or any convex superset of the quasi-convex polytope P and decide at run time whether a point of the set is member of the polytope or not.

For simplicity, our implementation [10] does not even compute the convex hull of P but its rectangular hull. In this simple situation, the lower (upper) bound of a loop at level r in $\mathcal{P}'$ is given by the minimum (maximum) of all lower (upper) bounds of the loops at level r of all program parts enumerating $P_1, \ldots, P_k$.

Future experiments will have to reveal whether the convex hull pays off compared to the less precise rectangular hull, since the price of scanning fewer points at run time is an increase in complexity for the computations in the loop bounds.

The body of the loop nest consists of one if clause per target polytope $P_{k'}$ $(1 \leq k' \leq k)$ which guards the execution of the statement associated with $P_{k'}$.

Remark Note that we place the if clause for a target polytope $P_{k'}$ into the outermost target loop of $\mathcal{P}'$ for which all dimensions of $P_{k'}$ are enumerated (see Figure 2).[1]

Extension to non-unit strides If at least one program part has a non-unit stride, we compute the gcd of the strides of all program parts in every dimension, to get the stride for the target loop in this dimension. In compensation, we must augment the guard preceding the loop body. For more details see [17].

[1]In special cases, e.g., in Figure 2, this could still be improved by hoisting the guard out of a loop with standard code motion techniques.

```
                                        P': for i := 1 to max(m,5)
P'_1: for i := 1 to m                          if (1 ≤ i ≤ m) then
         S_1                                      S_1
      end for                                  endif
                                               for j := i to i+3
P'_2 : for i := 2 to 5           →                if (2 ≤ i ≤ 5) then
          for j := i to i+3                          S_2
             S_2                                  endif
          endfor                               endfor
       endfor                               endfor
```

Figure 2. Merging two target program parts with the run-time solution

Summary The run-time solution is generally applicable, but the generated conditionals cause run-time overhead. They are necessary to check, for every iteration, in which target polytope it is contained and, thus, which statements must be executed. Due to the construction of the hull (regardless of whether it is convex or rectangular), it is even possible that no statement is executed. The next subsection describes an alternative to avoid this overhead.

4.2. Compile-time solution

To avoid the control overhead of the run-time solution, we divide the quasi-convex target polytope P into several proper polytopes at compile time, which can be transformed directly into loop nests.

More technically, the compile-time solution generates a *sequence* of loops for the first (outermost) dimension and, within every generated loop, proceeds recursively with the next dimension. Let us first give an intuitive description of our algorithm. In order to generate the sequence of loops for dimension r, the algorithm scans dimension r with a hyperplane $H_r(x)$ orthogonal to r and with coordinate x in dimension r, and reports every coordinate at which a target polytope $P_{k'}$ starts or ends. The starting point (end point) means of the scan is the first (last) point at which the intersection of $H_r(x)$ and $P_{k'}$ is not empty. More formally, $P_{k'}$ starts or ends at x iff

$(\forall x' : x' < x : H_r(x') \cap P_{k'} = \emptyset \wedge H_r(x) \cap P_{k'} \neq \emptyset)$ or
$(\forall x' : x' > x : H_r(x') \cap P_{k'} = \emptyset \wedge H_r(x) \cap P_{k'} \neq \emptyset)$
respectively.

Note that, in the presence of structure parameters, the start and end coordinates are not known at compile time.

Example 4.1
Consider the target program parts in Figure 3. (Of course, S_1 and S_2 may also contain further loops.) The merged target program for the case $30 \leq m \leq 50$ is given on the right.

```
P'_1:  for i := 10 to m                 P':  for i := 10 to 29
           S_1                                   S_1
       endfor                                endfor
                                             for i := 30 to m
                                    →            S_1
P'_2:  for i := 30 to 50                         S_2
           S_2                               endfor
       endfor                                for i := m+1 to 50
                                                 S_2
                                             endfor
```

Figure 3. Merging two program parts with the compile-time solution (for $30 \le m \le 50$)

The value of m determines whether the target polytopes do overlap and, if so, how far.

The only solution we see is to create target loops for every possible permutation of the parameters and, thus, for every possible permutation of the start and end points.

More precisely, for every dimension d, we compute different sequences of loops, each of which enumerates dimension d according to different permutations of the structure parameters. All these possible sequences of loops for dimension d are then combined as branches of a conditional whose guards test for the permutations of the structure parameters. At run time, we check which permutation of the parameters is actually given and, thus, which branch of the target program must be executed.

Technically, the scan of a dimension is realized by just taking all lower and upper bounds of the loops to be merged, and trying to sort them, which again leads to the difficulties with the unknown values of the parameters at compile time.

Complexity For N parameters in the source loop nest, we have $N!$, i.e., exponentially many permutations! The situation becomes even worse if we have maximum and minimum expressions in the loop bounds: in this case, we do not know at compile time which of the arguments will contribute to the maximum at a given index vector and, thus, we consider every argument of a maximum or minimum expression as a new structure parameter. Since this case occurs quite frequently in practice, the increase in the number of possible permutations is enormous. Note that this negative result is valid for any scheme which tries to avoid control overhead, e.g., [11].

Example 4.2
If we modify Example 4.1 such that all four loop bounds are parameters, we have 24 possible permutations (for the outermost loop), i.e., branches in the target program.

```
                                    forall i := 1 to n
for i := 1 to n                         A[i] := f(i);
    A[i] := f(i);               →   endforall
    B[i] := g(A[i], B[i-1]);        for i := 1 to n
endfor                                  B[i] := g(A[i], B[i-1]);
                                    endfor
```

Figure 4. Loop distribution as a result of compile-time solution

Extension to non-unit strides Since there is no way of merging different strides into a single loop without using conditionals, the solution for a non-unit stride is inevitably the same as in the run-time solution, thus causing conditionals even in the compile-time solution.

Generation of easily readable and efficient code It has been shown that many individual techniques for automatic parallelization, e.g., loop distribution, loop peeling, etc. can be expressed as a space-time mapping [15, 18]. However, the reverse problem has not been addressed so far: to check whether a given space-time mapping corresponds to loop distribution, loop peeling, etc. This knowledge would lead us directly to efficient target code. E.g., in our examples, the compile-time solution leads automatically to the desired target programs.

Example 4.3
Consider the source loop nest in Figure 4, left, and the schedule $t(\langle S_1,(i)\rangle) = 0$ and $t(\langle S_2,(i)\rangle) = i$. This space-time mapping can be interpreted as a (very simple) case of loop distribution and, indeed, the compile-time solution generates the same code as loop distribution for this case (Figure 4, right).

Similarly, our compile-time version automatically leads to loop peeling, if appropriate. E.g., for two one-dimensional program parts $\mathcal{P}_1$ and $\mathcal{P}_2$, which enumerate iterations $0, \cdots, n-1$ and $1, \cdots, n$, respectively, we obtain a target program with three parts: first, instance 0 of the body of $\mathcal{P}_1$, then a loop from 1 to $n-1$ containing both body statements, and finally instance n of the body of $\mathcal{P}_2$.

5. Conclusions

We have seen in Section 3 that we can extend code generation easily to singular transformation matrices. Therefore, the proposed code generation scheme can deal with any arbitrary affine schedule and allocation. This is a significant improvement over former code generation algorithms, especially since, in practice, some dimensions of schedule and allocation are quite often linearly dependent.

There is need for future work. In the extension of the basis of the transformation matrix (Section 3.2.2), we insert a unit vector for every missing row. It should be examined whether an insertion of a non-unit vector is better and, if so, which one is the best.

Section 4 proposes solutions for dealing with piecewise affine schedules or allocations for individual statements. The main challenge is to merge the individual target program parts. We have proposed two solutions.

The run-time solution (Section 4.1) creates a loop nest which, in general, enumerates a superset of the target index points to be executed; evaluating conditional expressions at run time guarantees the correctness of the target program. This method is generally applicable but causes run-time overhead.

The compile-time solution (Section 4.2) avoids this overhead as far as possible by dividing the index space of the quasi-convex target polytope into several parts. However, due to parameters in the input program and due to minimum or maximum expressions in the target program parts, the number of possible target programs grows exponentially. Even for very simple programs the results are frightening: for a source program of a few lines with three nested loops, we obtain 8 expressions for the loop bounds of the outermost dimension, which cause 40320 permutations, i.e., an if cascade with 40320 branches. We cancelled the code generation after 20 minutes, when the target program had grown to approximately 130 MB of disk space.

In addition, for non-unimodular transformations, even the compile-time solution leads to guards at run time, because we must select the proper stride.

All in all, the compile-time version looks more promising at first sight but, in practice, the run-time solution will probably be preferred, except for the simple situation in which there is no parameter in the bounds of the source loops.

Acknowledgements

This work has been funded by the German Research Foundation (DFG) under project RecuR and by the German Academic Exchange Service (DAAD) under its PROCOPE programme. We are grateful to Jean-François Collard to numerous discussions on code generation.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, 1991.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1993.

[3] Z. Chamski. Scanning polyhedra with DO loop sequences. In B. C. Sendov and I. Dimov, editors, *Proc. Workshop on Parallel Architectures (WPA'92)*. Elsevier (North-Holland), 1992.

[4] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G. 10.3*, pages 185–194. North-Holland, Apr. 1994.

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

[6] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. J. Parallel Programming*, 25(6):447–496, Dec. 1997.

[7] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, Oct. 1992.

[9] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer-Verlag, 1996.

[10] M. Griebl and C. Lengauer. The loop parallelizer LooPo—Announcement. In D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, Lecture Notes in Computer Science 1239, pages 603–604. Springer-Verlag, 1997.

[11] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, 1994.

[12] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[13] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Int. J. Parallel Programming*, 22(2):183–205, Apr. 1994.

[14] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.

[15] W. Pugh. Uniform techniques for loop optimization. In *Proc. Int. Conf. on Supercomputing (ICS'91)*, pages 341–352, 1991. ACM SIGARCH, ACM Press.

[16] A. Schrijver. *Theory of Linear and Integer Programming*. Series in Discrete Mathematics. John Wiley & Sons, 1986.

[17] S. Wetzel. Automatic code generation in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, Nov. 1995. http://www.fmi.uni-passau.de/loopo/doc/wetzel-d.ps.gz.

[18] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

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

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