Finding Synchronization-Free Parallelism for Non-uniform Loops

Volodymyr Beletskyy

Faculty of Computer Science, Technical University of Szczecin, Zolnierska 49 st.,
71-210 Szczecin, Poland,
vbeletskyy@wi.ps.pl

Abstract. A technique, permitting us to find synchronization-free parallelism in non-uniform loops, is presented. It is based on finding affine space partition mappings. The main advantage of this technique is that it allows us to form constraints for finding mappings directly in a linear form while known techniques result in building non-linear constraints which should next be linearized. After finding affine space partition mappings, well-known code generation approaches can be applied to expose loop parallelism. The technique is illustrated with two examples.

1 Introduction

A lot of transformations have been developed to expose parallelism in loops, minimize synchronization, and improve memory locality in the past [1],[3],[4],[6],[7],[8],[9],[10],[11],[12],[14],[15],[19]. However, there are the following questions. Which of these methods permit us to find synchronization-free parallelism and what is their complexity for non-uniform loops.

According to a study by Sass and Mutka[18], a majority of the loops in scientific code are imperfectly nested, and a majority of the performance-increasing techniques developed in the past assume that loops are perfectly nested, that is, imperfectly nested loops deserve more attention from the research community.

This paper presents a technique permitting us to find synchronization-free parallelism in non-uniform loops. We refer to a particular execution of a statement for a certain iteration of the loops, which surround this statement, as an operation. The operations of a loop are divided into partitions, such that dependent operations are placed in the same partition. A partitioning is described by an affine mapping for each loop statement.

An *m*-dimensional affine partition mapping for statement *s* in a loop is an *m*-dimensional affine expression $\bar{\phi}_s = C_s \bar{i} + \bar{c}_s$, which maps an instance of statement *s*, indexed by its iteration vector $\bar{i}$, to an *m*-dimensional vector. Given affine mappings, well-known techniques for generating parallel code can be applied, for example, [2],[4],[5],[17].

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2658, pp. 925–934, 2003.

2 Dependence Analysis

Our algorithm is based on the dependence analysis proposed by Pugh and Wonnacott [16]. That analysis permits us to extract exact dependence information for any single structured procedure in which the expressions in the subscripts, loop bounds, and conditionals are affine functions of the loop indices and loop-independent variables, and the loop steps are known constants. Dependences are presented with dependence relations. A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that stand for the values of the loop indices at the source and destination of the dependence and the values of the symbolic constants. A dependence relation is a tuple relation. An integer *k*-tuple is a point in Z^k. A tuple relation is a mapping from tuples to tuples.

The basic merits of the dependence analysis proposed by Pugh and Wonnacott are as follows: i) it is exact; ii) it is valid for both perfectly and imperfectly nested loops; iii) it permits value-based dependences to be calculated.

A dependence between operations $\overline{I}$ and $\overline{J}$, which are the source and destination of the dependence, respectively, is value-based if: $\overline{I}$ is executed before $\overline{J}$; $\overline{I}$ and $\overline{J}$ refer to a memory location *M*, and at least one of these references is a write; the memory location *M* is not written between operation $\overline{I}$ and operation $\overline{J}$.

The dependence analysis by Pugh and Wonnacott is implemented in Petit, a research tool for doing dependence analysis and program transformations. To carry out dependence analysis manually, the Omega calculator can be applied [13].

An affine loop nest is non-uniform if it originates non-uniform dependence relations represented by an affine function *f* that expresses the dependence sources $\overline{I}$ in terms of the dependence destinations $\overline{J}$ ($\overline{I} = f(\overline{J})$) or vice versa.

An algorithm proposed in this paper is applicable for those loops that meet the restrictions of the dependence analysis proposed by Pugh and Wonnacott [16].

3 Space-Partition Constraints

Our approach is applicable to the following imperfectly nested loop considered in [19]

$$
\begin{array}{ll}
 & \text{do } x_1 = L_1, U_1 \\
S_{1a}: & \quad H_{1a}(x_1) \\
 & \quad \text{do } x_2 = L_2, U_2 \\
S_{2a}: & \quad\quad H_{2a}(x_1, x_2) \\
 & \quad\quad \cdots \\
 & \quad\quad \text{do } x_n = L_n, U_n \\
S_n: & \quad\quad\quad H_n(x_1, \ldots, x_n) \\
 & \quad\quad\quad\quad \cdots
\end{array}
\tag{1}
$$

S_{2b}: $H_{2b}(x_1, x_2)$

S_{1b}: $H_{1b}(x_1)$,

where the loop bounds of x_k are affine constraints over surrounding loop variables $x_1, \ldots, x_{k-1}$ and some symbolic integer constants, or formally:

$$L_k = \max(l_{k,1}, l_{k,2}, \ldots)$$
$$U_k = \max(u_{k,1}, u_{k,2}, \ldots),$$

where

$$l_{k,p} = \lceil (l^0_{k,p} + l^1_{k,p} x_1 + \ldots + l^{k-1}_{k,p} x_{k-1}) / l^k_{k,p} \rceil$$

$$u_{k,p} = \lfloor (u^0_{k,p} + u^1_{k,p} x_1 + \ldots + u^{k-1}_{k,p} x_{k-1}) / u^k_{k,p} \rfloor$$

and all $l_{k,p}$ and $u_{k,p}$ are integer constants, except possibly for $l^0_{k,p}$ and $u^0_{k,p}$, which may be symbolic constraints but must still be loop invariants in the loop nest. The ceiling and floor functions are introduced to convert rationals to integers. In general, a lower(upper) loop bound is a maximum(minimum) of affine constraints with rational coefficients. This ensures that the space defined by any set of loops in the loop nest is a convex polyhedron.

In this section, we consider the following task. Given a set of dependences originated by a loop and presented with dependence relations

$$D = \{\bar{I}_j -> \bar{J}_j, j = 1,2,\ldots, q\},$$

find m_s-dimensional affine space partition mappings $\bar{\phi}_s = C_s \bar{i} + \bar{c}_s$ for each *s=1a, 2a, ..., n,...,2b, 1b,* such that $\bar{\phi}_{si}(\bar{I}_j) = \bar{\phi}_{sk}(\bar{J}_j) = \bar{P}_m$, where *si, sk* are the statements which instances originate the source and destination of the dependence $\bar{I}_j -> \bar{J}_j$, C_s is a matrix of dimensions m_s x *n,* $\bar{c}_s$ is an m_s-dimensional vector representing a constant term, $\bar{P}_m$ is a vector representing the identifier of a processor to execute the source and destination of the dependence $\bar{I}_j -> \bar{J}_j$.

Let $\bar{I}_j$, $\bar{J}_j$ be represented in the following form

$$\bar{I}_j = A_{1j} * \overline{i1}_j + \overline{B1}_j, \quad \bar{J}_j = A_{2j} * \overline{i2}_j + \overline{B2}_j,$$

where $\bar{I}_j$, $\bar{J}_j$ are $m1_j$ and $m2_j$-dimensional vectors, respectively, $m1_j <= n$, $m2_j <= n$, n is the number of the loop nests, $\overline{B1}_j$, $\overline{B2}_j$ are $m1_j$ and $m2_j$-dimensional vectors, respectively, $\overline{i1}_j$ and $\overline{i2}_j$ are n-dimensional vectors, A_{1j} and A_{2j} are matrices of dimensions $m1_j$ x n and $m2_j$ x n, respectively.

Let us write matrices A_{1j} and A_{2j} in the following form

$$A_{1j} = [\overline{A}^{1}_{1j} \ \overline{A}^{2}_{1j} \ \ldots \overline{A}^{n}_{1j}], \quad A_{2j} = [\overline{A}^{1}_{2j} \ \overline{A}^{2}_{2j} \ \ldots \overline{A}^{n}_{2j}],$$

where $\overline{A}^{i}_{1j}$ and $\overline{A}^{i}_{2j}$, $i=1,2,\ldots,n$ represent the columns of A_{1j} and A_{2j}, respectively.

If a dependence $\overline{I}_j -> \overline{J}_j, j \in [1, \ q]$ is the self dependence, that is, it is originated with the same statement s, we seek an affine space partition mapping $\overline{\phi}_s = C_s \overline{i} + \overline{c}_s$ such that the following condition is satisfied

$$C_s \overline{I}_j + \overline{c}_s = C_s \overline{J}_j + \overline{c}_s$$

or

$$C_s \overline{I}_j - C_s \overline{J}_j = 0,$$

which means that the same processor executes operations $\overline{I}_j$ and $\overline{J}_j$.

Let us rewrite the equation above as follows

$$C_s(\overline{A}^{1}_{1j} i1^{1}_{j} + \overline{A}^{2}_{1j} i1^{2}_{j} + \ldots + \overline{A}^{n}_{1j} i1^{n}_{j} + \overline{B1}_j) -$$

$$- C_s(\overline{A}^{1}_{2j} i2^{1}_{j} + \overline{A}^{2}_{2j} i2^{2}_{j} + \ldots + \overline{A}^{n}_{2j} i2^{n}_{j} + \overline{B2}_j) = 0,$$

where $i1^{1}_{j}, i1^{2}_{j}, \ldots, i1^{n}_{j}$ and $i2^{1}_{j}, i2^{2}_{j}, \ldots, i2^{n}_{j}$ are the coordinates of $\overline{i1}_j$ and $\overline{i2}_j$, respectively, and transform it to the form

$$\langle \overline{C}_s, \overline{A}^{1}_{1j} \rangle i1^{1}_{j} + \langle \overline{C}_s, \overline{A}^{2}_{1j} \rangle i1^{2}_{j} + \ldots + \langle \overline{C}_s, \overline{A}^{n}_{1j} \rangle i1^{n}_{j} + \langle \overline{C}_s, \overline{B1}_j \rangle -$$

$$- \langle \overline{C}_s, \overline{A}^{1}_{2j} \rangle i2^{1}_{j} - \langle \overline{C}_s, \overline{A}^{2}_{2j} \rangle i2^{2}_{j} - \ldots - \langle \overline{C}_s, \overline{A}^{n}_{2j} \rangle i2^{n}_{j} - \langle \overline{C}_s, \overline{B2}_j \rangle = 0, \tag{2}$$

where $\langle \overline{x}, \overline{y} \rangle$ denotes the inner product of vectors $\overline{x}$ and $\overline{y}$, $\overline{C}_s$ represents an arbitrary row of C_s.

If a dependence $\overline{I}_j -> \overline{J}_j, j \in [1, \ q]$ is originated with two different statements *s1* and *s2*, we seek two affine space partition mappings $\overline{\phi}_{s1} = \overline{C}_{s1} \overline{i} + \overline{c}_{s1}$ and $\overline{\phi}_{s2} = \overline{C}_{s2} \overline{i} + \overline{c}_{s2}$ such that the following condition is satisfied

$$C_{s1} \overline{I}_j + \overline{c}_{s1} = C_{s2} \overline{J}_j + \overline{c}_{s2}.$$

Let us rewrite the above condition as follows

$$C_{s1}(\overline{A}^{1}_{1j} i1^{1}_{j} + \overline{A}^{2}_{1j} i1^{2}_{j} + \ldots + \overline{A}^{n}_{1j} i1^{n}_{j} + \overline{B1}_j) + \overline{c}_{s1} =$$

$$C_{s2}(\overline{A}^{1}_{2j} i2^{1}_{j} + \overline{A}^{2}_{2j} i2^{2}_{j} + \ldots + \overline{A}^{n}_{2j} i2^{n}_{j} + \overline{B2}_j) + \overline{c}_{s2},$$

and transform it to the form

$$\langle \overline{C}_{s1}, \overline{A}^{1}_{1j} \rangle i1^{1}_{j} + \langle \overline{C}_{s1}, \overline{A}^{2}_{1j} \rangle i1^{2}_{j} + ... + \langle \overline{C}_{s1}, \overline{A}^{n}_{1j} \rangle i1^{n}_{j} + \langle \overline{C}_{s1}, \overline{B1}_{j} \rangle + c_{s1} -$$

$$- \langle \overline{C}_{s2}, \overline{A}^{1}_{2j} \rangle i2^{1}_{j} - \langle \overline{C}_{s2}, \overline{A}^{2}_{2j} \rangle i2^{2}_{j} - ... - \langle \overline{C}_{s2}, \overline{A}^{n}_{2j} \rangle i2^{n}_{j} - \langle \overline{C}_{s2}, \overline{B2}_{j} \rangle - c_{s2} = 0, \quad (3)$$

where $\overline{C}_{s1}, \overline{C}_{s2}$ represent an arbitrary row of C_{s1}, C_{s2}, respectively, c_{s1}, c_{s2} are unknown constant terms which are dependent on $\overline{C}_{s1}, \overline{C}_{s2}$.

Let us introduce an r_j-dimensional vector $\bar{i}_j$ which consists of all uncommon coordinates(having different names) of $\overline{I}_j, \overline{J}_j$ and the coordinates of this vector be $i^1_j, i^2_j, ..., i^{r_j}_j$, $r_j <= m1_j + m2_j$. Rewrite equations (2) and (3) in the following form

$$\sum_{k=1}^{r_j} D^k_j \; i^k_j + d_j = 0, \quad (4)$$

where D^k_j and d_j are formed as follows:

i) for self dependences, it is the sum of all those $\langle \overline{C}_{s1}, \overline{A}^{m}_{1j} \rangle$ and $-\langle \overline{C}_{s1}, \overline{A}^{p}_{2j} \rangle$ for which the following condition holds $i1^m_j = i2^p_j = i^k_j$; $d_j = \langle \overline{C}_{s1}, \overline{B1}_{j} \rangle - \langle \overline{C}_{s1}, \overline{B2}_{j} \rangle$;

ii) for dependences originated with two different statements, it is the sum of all those $\langle \overline{C}_{s1}, \overline{A}^{m}_{1j} \rangle$ and $-\langle \overline{C}_{s2}, \overline{A}^{p}_{2j} \rangle$ for which the following condition holds $i1^m_j = i2^p_j = i^k_j$; $d_j = \langle \overline{C}_{s1}, \overline{B1}_{j} \rangle - \langle \overline{C}_{s2}, \overline{B2}_{j} \rangle + c_{s1} - c_{s2}$.

Algorithm. Find affine space partition mappings for a loop originating the dependences defined by set *D*.

1. From each dependence $\overline{I}_j -> \overline{J}_j$, $j = 1,2,..., q$, build the constraint in the form of (4).
2. Construct a system of linear equations of the form

$$D^k_j = 0,$$
$$d_j = 0, \; j = 1,2,..., q, \; k = 1,2,..., r_j$$

which we rewrite as

$$A\bar{x} = 0,$$

where $\bar{x}$ is a vector representing all the unknown coordinates of $\overline{C}_s$ and constant terms c_s of the affine space partition mappings, *s=1a, 2a, ..., n,..., 2b, 1b*.

The remaining steps are the same as in the algorithm proposed in [15], namely

3. Eliminate all the unknowns c_s from $A\bar{x}=0$ with the Gaussian Elimination algorithm. Let the reduced system be $A'\bar{x}'=0$, where $\bar{x}'$ represents the unknown coordinates of $\bar{C}_s$.
4. Find the solution to $A'\bar{x}'=0$ as a set of basis vectors spanning the null space of A'.
5. Find one row of the desired affine partition mapping from each basic vector found in step 4. The coordinates of $\bar{C}_s$ are formed directly by the basic vector; the constant terms c_s are found from the coordinates of $\bar{C}_s$ using $A\bar{x}=0$.

After finding mappings, well-known techniques for generating parallel code can be applied, for example, [2],[4],[5],[17] and they are out of the scope of this paper.

4 Examples

Let us illustrate the technique presented by means of the two following examples.

Example 1:

```
for (i = 1; i <= n; i++)
  for (j = 1; j<= n; j++)
    for (k = 1; k <= n; k++)
          s1: a(j)=b(i);
```

For this loop, the dependences found with Petit are as follows

output *s1: a(j)* → *s1: a(j)*
{[i,j,k] → [i,j,k'] : 1 <= k < k' <= n && 1 <= i <= n && 1 <= j <= n},

output *s1: a(j)* → *s1: a(j)*
{[i,j,k] → [i',j,k'] : 1 <= i < i' <= n&&1 <= j <= n&&1 <= k<=n&&1<= k' <= n}.

We seek a mapping of the form $\bar{\phi}_{s1}=[C_{11} \quad C_{12} \quad C_{13}]\bar{i}$. According to our approach, we first form the following constraint

$$C_{11}*i+C_{12}*j+C_{13}*k=C_{11}*i+C_{12}*j+C_{13}*k'$$

$$C_{11}*i+C_{12}*j+C_{13}*k=C_{11}*i'+C_{12}*j+C_{13}*k',$$

which we simplify to the following form

$$C_{13}*(k-k')=0,$$
$$C_{11}(i-i')+C_{13}(k-k')=0.$$

The resulting constraint is as follows

$$C_{11} = 0$$
$$C_{13} = 0 .$$

The linearly independent solution to this system is

$$C_{11} = 0, C_{12} = 1, C_{13} = 0 .$$

Applying the Omega code generator (free available at ftp://ftp.cs.umd.edu/pub/omega) for the transformation of the source loop by means of the space partition mapping $C_1 = [0\ 1\ 0]$, we have got the following parallel code

```
parfor(p = 1; p <= n; p++)
  for(t1 = 1; t1 <= n; t1++)
    for(t2 = 1; t2 <= n; t2++)
      s1: a(p) = b(t1);
```

where `for` and `parfor` denote serial and parallel loops, respectively. The outer loop gives space partitioning while the inner loops define the statement instances executed serially by a given processor *p*.

Consider the following imperfectly nested loop.

Example 2:

```
for (i = 1; i <= n; i++){
   for (j = 1; j <= n; j++){
      for (k = 1; k <= n; k++){
         s1: c(i,j,k)=a(N-j,k);
      }
      s2: a(N-j+1,i)= b(j,k);
   }
}
```

This loop originates the following dependences found with Petit

anti *s1: a(N-j,k)* → *s2: a(N-j+1,i)*

{[i,j,i] → [i,j+1] : 1 <= i <= N && 1 <= j < N,},

anti *s1: a(N-j,k)* → *s2: a(N-j+1,i)*

{[i,j,k] → [k,j+1] : 1 <= i < k <= N && 1 <= j < N},

flow *s2: a(N-j+1,i)* → *s1: a(N-j,k)*

{[i,j] → [i',j-1,i] : 1 <= i < i' <= N && 2 <= j <= N}.

We seek mappings of the form $\bar{\phi}_{s1} = [C_{11} \quad C_{12} \quad C_{13}]\,\bar{i} + c_1$ and $\bar{\phi}_{s2} = [C_{21} \quad C_{22}]\,\bar{i} + c_2$. Firstly, we form the following constraint

$$C_{11}*i + C_{12}*j + C_{13}*i + c_1 = C_{21}*i + C_{22}*(j+1) + c_2$$

$$C_{11}*i + C_{12}*j + C_{13}*k + c_1 = C_{21}*k + C_{22}*(j+1) + c_2$$

$$C_{21}*i + C_{22}*j + c_2 = C_{11}*i' + C_{12}*(j-1) + C_{13}*i + c_1$$

and next transform it to the form

$$(C_{11} - C_{21} + C_{13})*i + (C_{12} - C_{22})*j + c_1 - c_2 - C_{22} = 0$$

$$C_{11}*i + (C_{12} - C_{22})*j + (C_{13} - C_{21})*k + c_1 - c_2 - C_{22} = 0$$

$$(C_{21} - C_{13})*i + (C_{22} - C_{12})*j - C_{11}*i' + c_2 - c_1 + C_{12} = 0.$$

On the basis of the equations above, we construct the following constraint

$$\begin{aligned} &C_{11} - C_{21} + C_{13} = 0 \\ &C_{12} - C_{22} = 0 \\ &c_1 - c_2 - C_{22} = 0 \\ &C_{11} = 0 \\ &C_{13} - C_{21} = 0 \\ &C_{21} - C_{13} = 0 \\ &c_2 - c_1 + C_{12} = 0. \end{aligned} \tag{5}$$

Eliminating c_1, c_2, we get

$$\begin{aligned} &C_{11} - C_{21} + C_{13} = 0 \\ &C_{12} - C_{22} = 0 \\ &C_{12} - C_{22} = 0 \\ &C_{11} = 0 \\ &C_{13} - C_{21} = 0 \\ &C_{21} - C_{13} = 0. \end{aligned}$$

The linearly independent solution to the system above is

$$C_{11} = 0, C_{12} = 1, C_{13} = 1, C_{21} = 1, C_{22} = 1.$$

From system (5) we find that $c_1 = 1$, $c_2 = 0$.

Applying the Omega code generator for the transformation of the source loop by means of the space partition mappings found, we have got the following parallel code

```
parfor(p=2; p<= 2*N+1; p++) {
  for(t1=1; t1<= N; t1++) {
    for(t2=max(-N+p-1,1); t2<=min(-t1+p-1,N); t2++) {
```

```
        s1(t1,t2,p-t2-1);
      }
      if (t1 >= 2 && t1 <= p-1 && t1 >= -N+p) {
        s1(t1,p-t1,t1-1);
      }
      if (t1 >= 2 && t1 <= p-1 && t1 >= -N+p) {
        s2(t1,p-t1);
      }
      for(t2 = max(-t1+p+1,1); t2 <= min(p-2,N); t2++) {
        s1(t1,t2,p-t2-1);
      }
      if (p <= N+1 && t1 <= 1) {
        s2(1,p-1);
      }
    }
  }
```

where `for` and `parfor` denote serial and parallel loops, respectively; *s1* and *s2* are the statements of the source loop.

The outer loop gives space partitioning while the inner loops define the statement instances which should be executed serially by a given processor *p*.

5 Related Work and Conclusion

Unimodular loop transformations[3],[19], permitting the outer loop to be parallelized, find synchronization-free partitions. But unimodular transformations do not allow such transformations as loop fission, fusion, scaling, reindexing, or reordering.

Techniques presented in [1],[11] enable finding synchronization-free partitioning only for perfectly nested loops, supposing statements within each loop iteration are indivisible.

The affine partitioning framework, considered in many papers, for example, [8],[9],[10],[15], unifies a large number of previously proposed loop transformations. It is the most powerful framework for the loop parallelization today allowing us to parallelize loops with both uniform and non-uniform dependences.

Work [15] is most closely related to ours. In contrast to that work and other known approaches, our technique permits us to form constraints for finding affine space partition mappings for non-uniform loops directly in a linear form without the necessity of applying the Farkas lemma to linearize the constraint, and hence it is less time-consuming than that of work [15] and other known approaches.

In the future research, we plan to extend our technique to find affine time partition mappings for the non-uniform loops which do not allow synchronization-free parallelization.

References

[1] Amarasinghe, S.P., Lam, M.S.: Communication optimization and code generation for distributed memory machines. In: Proceedings of the SIGPLAN'93 (1993) 126–138
[2] Ancourt, C., Irigoin, F.: Scanning polyhedra with do loops. In: Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press (1991) 39–50
[3] Banerjee, U.: Unimodular transformations of double loops. In: Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing (1990) 192–219
[4] Boulet, P., Darte, A., Silber, G.A., Vivien, F.: Loop paralellelization algorithms: from parallelism extraction to code generation. Technical report (1997)
[5] Collard, J.F., Feautrier, P., Risset, T.: Construction of do loops from systems of afffne constraints. Technical Report 93–15, LIP, Lyon (1993)
[6] Darte, A., Risset, T., Robert, Y.: Loop nest scheduling and transformations. In Dongarra, J., Tourancheau, B., eds.: Enviroments and tools for parallel science computing. North Holland (1993)
[7] Darte, A., Silber, G., Vivien, F.: Combining retiming and scheduling techniques for loop parallelization and loop tiling. Technical Report 96–34, Laboratoire de l'Informatique du Parallelisme (1996)
[8] Feautrier, P.: Some efficient solutions to the affine scheduling problem, part i, one dimensional time. International Journal of Parallel Programming 21 (1992) 313–348
[9] Feautrier, P.: Some efficient solutions to the affine scheduling problem, part ii, multidimensional time. International Journal of Parallel Programming 21 (1992) 389–420
[10] Feautrier, P.: Toward automatic distribution. Journal of Parallel Processing Letters 4 (1994) 233–244
[11] Huang, C., Sadayappan, P.: Communication-free hyperplane partitioning of nested loops. Journal of Parallel and Distributed Computing 19 (1993) 90–102
[12] Kelly, W., Pugh, W.: A framework for unifying reordering transformations. Technical Report CS-TR-2995.1, University of Maryland (1993)
[13] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Technical Report CS-TR-3445, University of Maryland (1995)
[14] Lim, W., Lam, M.S.: Communication-free parallelization via affine transformations. In: Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing (1994) 92–106
[15] Lim, W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1997)
[16] Pugh, W., D.Wonnacott: An exact method for analysis of value-based array data dependences. In: Workshop on Languages and Compilers for Parallel Computing (1993)
[17] Quillere, F., Rajopadhye, S., Wilde, D.: Generation of efficient nested loops from polyhedra. International Journal of Parallel Programming 28 (2000)
[18] Sass, R., Mutka, M.W.: Enabling Unimodular transformations. In: Proceedings of Supercomputing'94 (1994) 753–762
[19] Wolf, M.E.: Improving locality and parallelism in nested loops. Ph.D. Dissertation CSL-TR-92-538, Stanford University, Dept. Computer Science (1992)