Loop Parallelization: Revisiting Framework of Unimodular Transformations

Jordi Torres, Eduard Ayguadé, Jesus Labarta and Mateo Valero
Computer Architecture Department, Universitat Politècnica de Catalunya
Gran Capità s/n, Mòdul D6, 08034 - Barcelona, Spain
{torres | eduard | jesus | mateo}@ac.upc.es

Abstract

This paper extends the framework of linear loop transformations adding a new non-lineal step at the transformation process. The current framework of linear loop transformation cannot identify a significant fraction of parallelism. For this reason, we present a method to complement it with some basic transformations in order to extract the maximum loop parallelism in perfect nested loops with tight recurrences in the dependence graph. The parallelizing algorithm solves the important problem of deciding the set of transformations to apply in order to maximize the degree of parallelism, the number of parallel loops, within a loop nest, and presents a way of generating efficient transformed code that exploits coarse-grain parallelism on a MIMD system.

1: Introduction

Loop transformations have been recognized as one of the most important components of the parallelizing and vectorizing technology for current supercomputers. The aim is to transform nested-loop structures from the source program into semantically equivalent versions with more opportunities to parallelize them and to generate code that exploit efficiently the hardware resources of the architecture [10,20, 5].

Most of the existing compilers apply a set of basic loop transformations one at a time. At each step, it has to be decided whether the application of a transformation is legal and beneficial. Other approaches have been used, such as deciding a priory the composition of basic transformations or exploring the different compositions of them. These solutions are in general time consuming in obtaining the transformed code and analyzing their effects is also difficult.

An alternative solution to the problem is the use of linear transformations to specify a wide range of basic loop transformations (including loop interchange, loop reversal, loop skewing [4] and loop scaling [9]). In fact, any linear transformation modelled with a non-singular transformation matrix can be seen as a composition or product of these four basic transformations. The problem can be stated as finding a linear transformation that maximizes an objective function, such as degree of parallelism that can be obtained out of the loop [19]. The extend of application is restricted to perfectly nested loops.

From the specification of the source loop and the transformation matrix, a target loop has to be generated. This step has been solved in [4,19] when unimodular matrices are used. Additional problems appear when non-unimodular matrices are considered. The key point in the solutions proposed [9,7,22] is the use of the Fourier-Motzkin elimination method and the Hermite Normal Form decomposition.

Traditionally such a transformation applies to the whole loop. Recently, it has been argued that it can be profitable to apply different transformations to different statements in a loop [2, 16, 6, 8].

In [2] the inclusion of the statement dimension as a new component in the framework of loop transformations was proposed. It can be seen as an alignment before the transformation. In this paper we propose how to include a new step between the alignment and the linear transformation in order to exploit coarse-grain parallelism on a MIMD system. This new step is based on the ideas proposed by Banerjee with the Remainder Transformation [5]. We obtain automatically the transformation from the dependence graph that extracts the maximum loop parallelism. This work can be extended by applying a different unimodular transformation and alignment to each statement in the loop body [8].

The rest of the paper is organized as follows. In section 2 we present the terminology and assumptions used with this paper. In section 3 we show the framework of the method. In section 4 we explain the motivation for this new transformation by showing the limitations of the framework of linear loop transformation. In section 5 we formalize the method proposed. Finally, in section 6 we briefly discuss the results obtained for the working example and present the conclusions and future work.

2: Terminology and assumptions

Through out this paper we consider perfectly nested loops $\{L_1, ..., L_n\}$ where the lower and upper bounds (l_k and u_k, respectively) for any loop L_k ($1 \le k \le n$) are affine functions of indices of its outer loops $L_1, ..., L_{k-1}$, that is,

$$i_k \ge a_{k,0} + a_{k,1} \cdot i_1 + ... + a_{k,(k-1)} \cdot i_{k-1} = l_k$$
$$i_k \le b_{k,0} + b_{k,1} \cdot i_1 + ... + b_{k,(k-1)} \cdot i_{k-1} = u_k$$

The iteration space for this loop nest is defined as

$$IS = \{ (i_1, ..., i_n) \in Z^n, l_k \le i_k \le u_k, 1 \le k \le n\}$$

and can be written following a matrix notation

$$\alpha \cdot I \le \beta$$

where

$$\alpha = \begin{bmatrix} L - ID \\ ID - U \end{bmatrix} \qquad \beta = \begin{bmatrix} -l \\ u \end{bmatrix}$$

L is constructed from the coefficients $a_{k,i}$ ($1 \le k \le n$, $1 \le i \le k-1$) of the loop indices in the lower bound expressions, U is constructed from the coefficients $b_{k,i}$ ($1 \le k \le n$, $1 \le i \le k-1$) of the loop indices in the upper bound expressions, ID is the identity matrix, and l and u are constructed from the independent coefficients $a_{k,0}$ and $b_{k,0}$ ($1 \le k \le n$) of the lower and upper bounds respectively.

For example, for the loop nest in 1.a, the IS of the loop can be expressed in the following way

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \le \begin{bmatrix} 0 \\ 0 \\ 9 \\ 9 \end{bmatrix}$$

Figure 1.b shows the aspect of the IS for this loop. Each point represents the execution of one iteration of the inner loop body.

In the scope of this paper we consider dense iteration spaces, i.e., spaces where all points correspond to iterations of the loop.

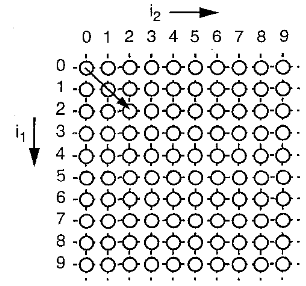
The loop body is composed of multiple assignment statements $\{S_1, ..., S_m\}$ that reference array variables whose subscripts are affine functions of loop indices $i_1, ..., i_n$. Let V be the set of statements in the loop body. The *Statement per Iteration Space* SIS of a loop nest is defined as the cartesian product

$$SIS = IS \times V$$

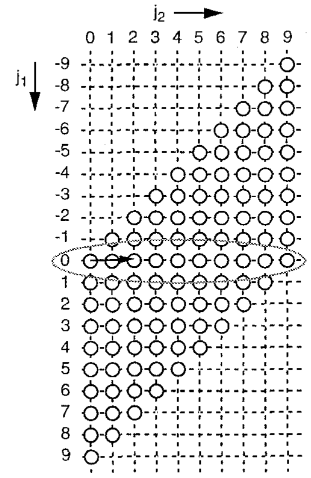
Each point in the SIS represents the execution of an iteration of a statement of the loop body. Dependence relations between a pair of statements S_i and S_j appear when there is an execution ordering between them [3]. We do not distinguish between the different kinds of data dependences because they all impose ordering constraints in the same way. When dependence relations are uniform (i.e., invariant through the SIS), they can be characterized by distance vectors $\overline{d}=<d^1, ..., d^n>$ expressing the number of iterations that the dependence extends across in each loop dimension. In a sequential loop nest, the iterations are executed in lexicographic order; thus, dependences extracted from such a source program can always be represented as a set of lexicographically positive vectors. A vector $\overline{d}$ is lexicographically positive, written $\overline{d} > \overline{0}$, if $\exists i$:($d^i>0$ and $\forall$ $j<i$:$d^j \ge 0$).

```
DO i1= 0,9
 DO i2= 0,9
  A[i1,i2]= A[i1-2,i2-2]* 2
 ENDO
ENDO
```

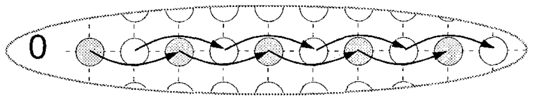
(a)



(b)



(c)



(d)

Figure 1: (a) Code of the working example, (b) representation of the original IS and (c) representation of the target IS after transformation and (d) detail of row 0.

Let E be the set of dependence relations in the loop. When uniform dependences are considered, the dependence graph G(V, E) is used to summarize all the dependence information. In this graph, vertices represent statements of the loop body and edges represent dependence relations between them. In this paper, all dependences are assumed to be uniform. We denote the dependence relations between any pair of statements S_i and S_j with $\bar{d}_{ij}$.

A *chain* C_{ij} is an ordered set of arcs $C_{ij} = \{\bar{d}_{ik}, \bar{d}_{kl}, ..., \bar{d}_{mj}\}$ between two statements S_i and S_j such that each node in the chain is visited only once. Given a chain C_{ij}, we define its *weight* w_{ij} as

$$w_{ij} = \sum_{d_{lm} \in C_{ij}} d_{lm}$$

This weight w_{ij} represents the number of iterations between any pair of instances of statements S_i and S_j depend from each other through chain C_{ij}.

A *recurrence* R is a cycle or closed chain in the dependence graph. A *hamiltonian recurrence* is a recurrence going through all nodes in the dependence graph.

Let $B = \{ R_1, R_2, ..., R_r \}$ be the set of recurrences in a given dependence graph G. This graph G is an *acyclic dependence graph* when $B=\emptyset$ and it is a *cyclic dependence graph* when $|B| \geq 1$. When at least one recurrence of B is hamiltonian, the graph is called *hamiltonian graph.*

3: Framework

In this section we briefly review the framework of linear loop transformation where our proposal is developed. A loop transformation is a mapping between two iteration spaces (named original and target IS). In this paper we consider linear transformations modelled by unimodular matrices (i.e., matrices whose determinant is ±1). These transformations can be used to model some basic transformations such as permutations, skewing and reversal [4,19].

Let I be a point of the original IS, J a point of the target IS and T the transformation matrix. The relationship between them is

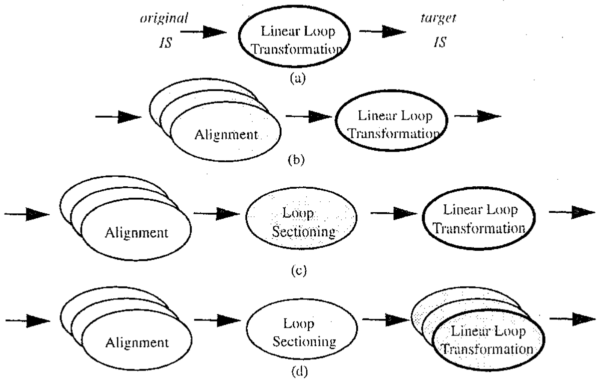
$$J = T \cdot I$$

Let $\bar{d}$ be a distance vector in the original IS. Since T is a linear transformation, $T \cdot \bar{d}$ is the transformed distance vector in the target IS. A transformation T is legal if

$$T \cdot \bar{d} > \bar{0}$$

for all dependence relations $\bar{d}$ in the dependence graph G(V, E). This means that each transformed dependence has to be lexicographically positive in the target IS.

This basic loop transformation can be extended including the statement dimension as a new component. In [2] it has been shown how different displacements for each statement can be used to break dependences. In [16] this technique has been applied to eliminate non-local references. Figure 2.b summarizes the procedure when the alignment step is included in the transformation process.



original IS
Linear Loop Transformation
target IS
(a)
Alignment
Linear Loop Transformation
(b)
Alignment
Loop Sectioning
Linear Loop Transformation
(c)
Alignment
Loop Sectioning
Linear Loop Transformation
(d)

Figure 2: Transforming a IS using a different approaches: (a) [4, 19], (b) [2,16], (c) this proposal, and (d) [8].

But as we will show in this paper, this transformation is not enough to identify a significant fraction of parallelism in a loop. We propose a way to solve this problem by adding a new non-lineal step at the transformation process based on the Remainder Transformation introduced by Banerjee [5]. This step called *loop sectionning* is introduced between the alignment and the linear transformation steps as shown in Figure 2.c.

Finally, it can be profitable to apply different unimodular transformations to different statements in the loop body. This transformation is named *multi-transformation.* In [8] this step is adressed and show how to avoid generating guards in most of the cases. Figure 2.d summarizes how multi-transformations can be combined with the previous steps. Hence we concentrate on the loop sectionning step in this paper.

4: Motivation for a new transformation

Linear loop transformations are a successful solution to the loop transformation problem, but they are not enough as we illustrate by the simple example shown in Figure 1.a (taken from [14]). Figure 1.b shows the original IS and Figure 1.c shows the target IS when the following transformation matrix is used

$$T = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

This nested loop contains only one dependence vector d=<2,2>. This dependence in the target space is transformed into d'=T·d=<0,2> as shown in the target space of Figure 1.c. This dependence is now carried by the innermost loop and thus the outer loop can be parallelized.

The code that scans the target IS is shown in Figure 3. If the outer loop is executed in parallel, the number of parallel tasks obtained is 19. However, if we carefully analyze each row in Figure 1.c, we can distinguish two independent subsets of iterations. Figure 1.d details the row corresponding to iteration j_1=0. Dependences reveal two independent subset of iterations. Thus we can conclude that linear transformations are not enough if this parallelism has to be exploited.

```
DO j1= -9,9
  DO j2= max(0,-j1),min(9,9-j1)
    A[j1+j2,j2]= A[j1+j2-2,j2-2]* 2
  ENDO
ENDO
```

Figure 3: Target code of the working example.

In this paper we propose a way of extracting this parallelism by adding a new non-linear step at the framework of lineal transformations based on the Remainder Transformations introduced by Banerjee [5]. Basically it consists in creating a new auxiliary space from the original IS where it is more easy to identify this parallelism and where it is possible to extract the maximum parallelism by a linear transformation. In the case of our working example, this auxiliary space can be obtained sectioning the innermost loop with strips of size 2 and then piling the sections as shown in Figure 4.b.

Since the remainder transformation adds an extra loop, it has an effect on the data dependence relations in the target loop. It adds a component to the distance vector. In this working example the new dependence vector becomes d=<2,0,1>. The code that scans this auxiliar iteration space is shown in Figure 5.

Thus, if we use the transformation matrix

$$T = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

to transform this auxiliar IS, we obtain the code shown in Figure 6.a where the 2 outermost loops can be parallelized because after the transformation the dependence becomes d=<0,0,1> and it is carried out by the innermost loop as one can see in the Figure 6.b

With this simple example we have tried to show that the linear loop transformation is not sufficient to extract full parallelism out of loops. In the following section we present our approach in order to exploit coarse-grain parallelism for nested loops on MIMD systems.

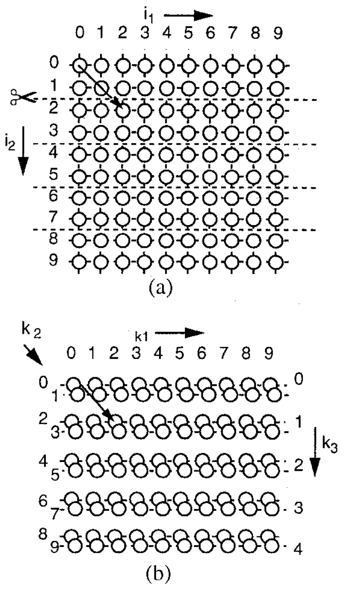


Figure 4: (a) original IS and (b) auxiliary IS of the working example.

```
DO k1= 0,9
  DO k2= 0,1
    DO k3= 0,4
      A[k1,k2+2·k3]= A[k1-2,k2+2·k3-2]* 2
    ENDO
  ENDO
ENDO
```

Figure 5: Code of the auxiliar IS.

```
DO j1= -8,9
  DO j2= 0,1
    DO j3= max(⌈-j1/2⌉, 0),min(⌊(9-j1)/2⌋, 4)
      A[j1+2·j3,j2+2·j3]= A[j1+2·j3-2,j2+2·j3-2
    ENDO
  ENDO
ENDO
```

(a)

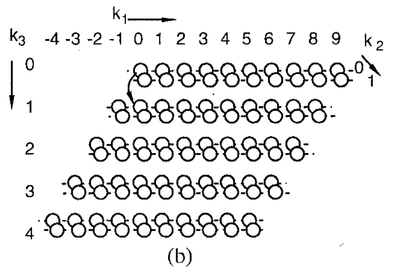


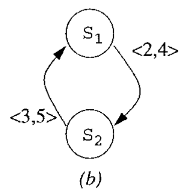
Figure 6: (a) Target code obtained and (b) target IS of working example.

5: Loop Sectioning

The work presented in this section focuses in loops that present tight recurrences in its dependence graph. In general, loops whose statements are involved in a dependence cycle are considered to be serial. However, techniques such as cycle shrinking [11] can be used to extract parallelism that may be present in the loop. In order to present the methodology proposed in this paper, consider the example of Figure 7.a slightly modified from [11] that has been used by several authors [12,15] in order to show their improvements with respect to the original cycle shrinking method.

```
   DO i1= 0,N1
    DO i2= 0,N2
S1:    A[i1,i2]= B[i1-3,i2-5]
S2:    B[i1,i2]= A[i1-2,i2-4]
    ENDO
   ENDO
```

(a)



(b)

Figure 7: (a) Nested Loops and (b) its dependence graph.

In this example, variable $A[i_1,i_2]$ is produced in statement $S_1(i_1,i_2)$ and consumed in statement $S_2(i_1+2,i_2+4)$ so there is a dependence between S_1 and S_2 with distance <2,4>. Similarly, there is a dependence between S_2 and S_1 with distance <3,5>. Both dependences are uniform. Figure 7.b shows a graphical representation of the dependence graph. In this example, there is a hamiltonian recurrence than involves all the statements of the loop body.

5.1: Loop Alignment

The basic idea of the alignment component that is added to each statement is to reduce the number of cross-iteration dependences. Figure 8 is intended to visually demonstrate our parallelizing algorithm. Consider that each statement is represented with a hyperplane in the SIS (as shown in Figure 8.a) and then we apply a different transformation to each statement of the loop in Figure 7.a in such a way that the resulting target IS is the one shown in Figure 8.c. We apply a shift of <2,4> to the hyperplane associated to statement S_1 and a null shift to the hyperplane associated to S_2. Different kinds of shades are used to identify the different hyperplanes. Notice that dependence <2,4> is now embedded in the sequential execution of each iteration because the hyperplanes of the statements have been shifted in order to make two dependent points be executed in the same iteration. Now the IS for S_1 is $[0..N_1:0..N_2]$ while for S_2 $[2..N_1+2,4..N_2+4]$. As a consequence, most of the iterations in the target loop execute both statements but others just execute one of them. The new bounds of the target loop nest have to be the union of the bounds for each statement: $[0..N_1+2, 0..N_2+4]$.

Let I be a point of the original hyperplane x corresponding to the statement S_x and J a point of the target hyperplane x after the alignment. The coordinates of each point $J=(j_1,...,j_m,...,j_n)$ can be obtained as

$$j_m = i_m + w^m_{xs}$$

where w^m_{xs} represent the weight of the hamiltonian recurrence from the statement S_x to the last statement S_s of the loop body in dimension m

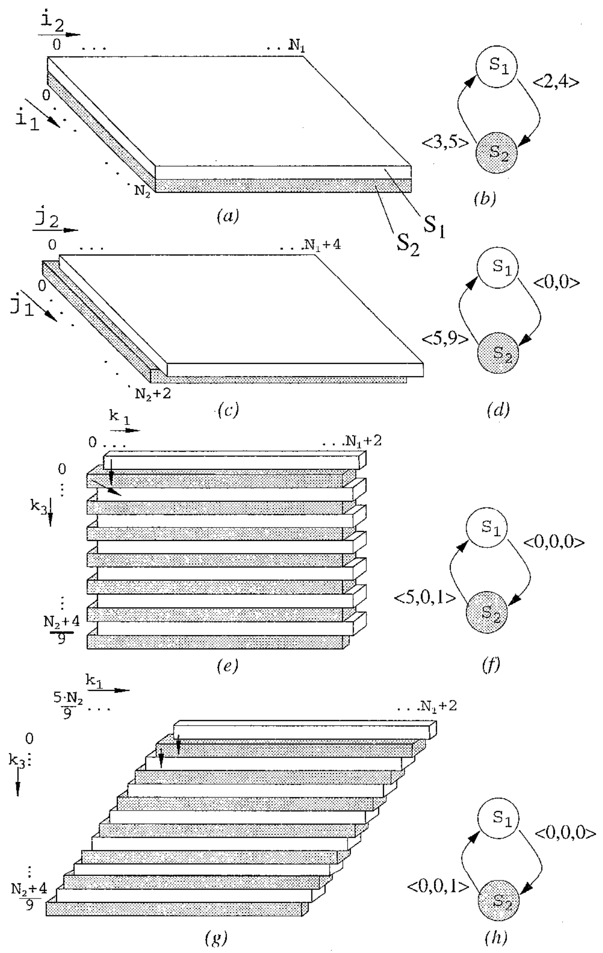


Figure 8: (a) Original SIS for the example of Figure 7, (b) original dependence graph, (c) auxiliar aligned SIS and (d) aligned dependence graph, (e) SIS after sectioning, (f) the resulting dependence graph, (g) SIS after transformation using matrix T and (h) the resulting dependence graph in the transformed space.

5.2: Loop Sectioning

Next we present a new step in the transformation process that is based on the Remainder Transformation introduced by Banerjee [5]. The IS defined by the original loop nest is divided into strips of some maximum size. Graphically it can be seen as if we section the first dimension i_1 of the iteration space in Figure 8.c in strips of size 9 and then pile them. The size of the strip that we propose is the weight of the hamiltonian recurrence in the dimension corresponding to the sectioned loop.

Loop sectioning is always legal; however it affects the data dependence relations in the loop. As we showed in the previous section, loop sectioning adds a new loop in the nest, and an element to the distance vectors. When a loop L_k is sectioned in two loops L_k and L_{k+1}, a dependence relation with a component d^k in the distance vector (corresponding to L_k), produces one or two dependence relations. If d^k is a multiple of the strip size ss, then the elements $d^k=0$ and $d^{k+1}=d^k/ss$ [21]. If d^k is not a multiple of ss, then the distance vector is transformed into two dependence relations, with the elements $<d^k,d^{k+1}>$ of the distance vectors set to $<-d^k \bmod ss, \lceil d^k/ss \rceil>$ and $<d^k \bmod ss, \lfloor d^k/ss \rfloor>$.

In our example the strip size is equal to the corresponding component in the distance vector. That means that the component d^k of the distance vector corresponding to the loop sectioned become equal 0 and the new component d^{k+1} becomes 1 (d^k=9, ss=9). This can be reflected in the dependence graph. Figure 8.f represents the dependence graph for the working example where the dependence <5,9> is transformed to <5,0,1>.

Let I be a point of the original space and J a point of the target space after loop sectioning. The coordinates of each point $J=(j_1,\ldots,j_m,\ldots,j_{n+1})$ can be obtained as

$$j_m = \begin{cases} i_m & m<k \\ i_m \bmod ss & m=k \\ \left\lfloor \dfrac{i_k}{ss} \right\rfloor & m=k+1 \\ i_{m+1} & m>k+1 \end{cases}$$

being k the sectioned loop and ss the value of the strip. Sectioning can be applied to any dimension of the original IS; however, [18] shows criteria to select the appropriate dimension. The value of ss is set to the weight of the recurrence in the component k corresponding to the sectioned loop. We denote this strip component with w_R^k.

5.3: Linear Loop transformation

Now we have an iteration space with a cross-iteration dependence with distance equal to <5,0,1>. This dependence is carried by the outermost loop. If we could transform this dependence into <0,0,1>, this dependence would be carried by the innermost loop and the two outermost loops would become parallel. This is what we can achieve with a linear loop transformation using the transformation matrix

$$T = \begin{bmatrix} 1 & 0 & -5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

applied to the auxiliary iteration space of Figure 8.e.

Graphically this transformation is represented in Figure 8.g. The arrows represent the two dependences in the graph of Figure 8.h. Figure 8.g shows the target space after apply the transformation. Observe that in this new space the dependence from S_2 to S_1 is transformed into <0,0,1> as shown in the dependence graph of Figure 8.h. Now in this target space, the iterations of the two outermost loops can be executed fully in parallel because one dependence (d_{12}=<0,0,0>) is internal to a iteration, and the other (d_{21}=<0,0,1>) is carried by the innermost loop.

This transformation can be expressed as a matrix and also as

$$j_m = \begin{cases} i_k & m=k \\ i_{k+1} & m=k+1 \\ i_m - i_{k+1} \times w_R^m & m \neq k \text{ and } m \neq k+1 \end{cases}$$

where we consider again that I represents a point of the original space and J a point of the transformed space.

5.4: Full Transformation

The framework presented in this paper decides which transformation to apply from the dependence graph.The transformation that we propose is obtained from the composition of the three previous transformations. Let I be a point of the original hyperplane x corresponding to the statement S_x and J a point of the target hyperplane x after the whole transformation. The coordinates of each point $J=(j_1,\ldots,j_m,\ldots,j_{n+1})$ can be obtained as

$$j_m = \begin{cases} i_m + w_{xs}^m - \left\lfloor \dfrac{i_k + w_{xs}^k}{w_R^k} \right\rfloor \times w_R^m & m<k \\ \left(i_k + w_{xs}^k\right) \bmod w_R^k & m=k \\ \left\lfloor \dfrac{i_k + w_{xs}^k}{w_R^k} \right\rfloor & m=k+1 \\ i_{m+1} + w_{xs}^{m+1} - \left\lfloor \dfrac{i_k + w_{xs}^k}{w_R^k} \right\rfloor \times w_R^{m+1} & m>k+1 \end{cases}$$

being k the sectioned loop. But the component j_k can be expressed in a different way. From the equality

$$\left(i_k + w^k_{js}\right) \bmod w^k_R = i_k + w^k_{js} - \left\lfloor \frac{i_k + w^k_{js}}{w^k_R} \right\rfloor \times w^k_R$$

the previous expression can be rewritten as follows

$$j_m = \begin{cases} i_m + w^m_{xs} - \left\lfloor \dfrac{i_k + w^k_{xs}}{w^k_R} \right\rfloor \times w^m_R & m < k+1 \\ \left\lfloor \dfrac{i_k + w^k_{xs}}{w^k_R} \right\rfloor & m = k+1 \\ i_{m+1} + w^{m+1}_{xs} - \left\lfloor \dfrac{i_k + w^k_{xs}}{w^k_R} \right\rfloor \times w^{m+1}_R & m > k+1 \end{cases}$$

In [18] we stated and proof several theorems that show the property of achieving the maximum parallelism that we have been referred along the previous section about the transformation proposed. In particular, it is important to remark that all the dependences used for the alignment components and strip component will be embedded in the sequential execution of the innermost loop, leading to fully parallel outermost loops.

The bounds of the transformed space are obtained from the union of the bounds of the original IS of each statement. In [17] we show how the bounds for each statement can be obtained. In general there are points of this transformed space where not all the statements of the loop body are executed. This can be controlled by guard conditionals. Unfortunately this code can be very expensive in terms of run-time overhead. In order to make this code less run-time expensive we define the core part as the part of the transformed space where all the statements of the loop body are executed. This core part can be executed without guard conditionals at the statement level, reducing the run-time overhead introduced by them. The code obtained has three parts, namely prolog, core and epilog parts. We have to include guards conditionals in the prolog and epilog parts in order to control the execution of each statement. The code that is obtained for the example is shown in Figure 9.

We assumed that there was only one recurrence. If more than one recurrence appears in the graph, it is necessary to use synchronization mechanisms that explicitly synchronize dependences not preserved when the parallel code obtained is executed. Explicit synchronization must be introduced for any edge not included in the hamiltonian recurrence in the graph going from node S_i to node S_j. Many mechanisms can be used to perform this synchronization. We will use counting semaphores as synchronization objects. As coupling between tasks will be very tight, we need a fast implementation of primitives on semaphores. The statement S_i source of the dependence will signal the end of its execution to the statement S_j sink of the same dependence, in order to allow its execution. During the code generation phase it is necessary to know the relationship of the parallel iterations related by a dependence d_{ij} in order to parametrize the semaphores. The reader can refer to [18] in order to find how to insert syncronization primitives.

```
DOALL $j_1 = \min(\lceil 2-(N_2+4)\cdot 5/9\rceil, \lceil -N_2\cdot 5/9\rceil), N_1+2$
   DOALL $j_2 = \max(\lceil 4+(j_1-N_1-2)\cdot 9/5\rceil, \lceil (j_1-N_1)\cdot 9/5\rceil, 0),$
              $\min(\lfloor N_2+4+(j_1-2)\cdot 9/5\rfloor, \lfloor N_2+9\cdot j_1/5\rfloor, 8)$

   $lower^1=\max(\lceil (2-j_1)/5\rceil, \lceil (4-j_2)/9\rceil)$
   $lower^2=\max(\lceil -j_1/5\rceil, \lceil -j_2/9\rceil)$
   $lower^{core}=\max(lower^1, lower^2)$
   $lower^{cover}=\min(lower^1, lower^2)$
   $upper^1=\min(\lfloor (N_1+2-j_1)/5\rfloor, \lfloor (N_2+4-j_2)/9\rfloor$
   $upper^2=\min(\lfloor (N_1-j_1)/5\rfloor, \lfloor (N_2-j_2)/9\rfloor)$
   $upper^{core}=\min(upper^1, upper^2)$
   $upper^{cover}=\max(upper^1, upper^2)$                     PROLOG

   if $(lower^2 \leq upper^{core}-1)$ and $(lower^2 \leq upper^2)$ then
      $B[j_1+5\cdot lower^2, j_2+9\cdot lower^2]=A[j_1+5\cdot lower^2-2$
                                          $, j_2+9\cdot lower^2-4]$
                                                    CORE
   if $(lower^{core} \leq upper^{core})$
      DO $j_3 = lower^{core}, upper^{core}$
         $A[j_1+5\cdot j_3-2, j_2+9\cdot j_3-4]=B[j_1+5\cdot j_3-5, j_2+9\cdot j_3-9]$
         $B[j_1+5\cdot j_3, j_2+9\cdot j_3]=A[j_1+5\cdot j_3-2, j_2+9\cdot j_3-4]$
      ENDO
      $LOWER=upper^{core}+1$
   else $LOWER=lower^{core}$
                                                    EPILOG
   if $(LOWER \leq upper^1)$ then
      $A[j_1+5\cdot upper^1-2, j_2+9\cdot upper^1-4]=B[j_1+5\cdot upper^1-5$
                                          $, j_2+9\cdot upper^1-9]$
   ENDOALL
ENDOALL
```

Figure 9: Transformed code obtained for the example of Figure 7.

6: Conclusions and Future Work

Program transformations are a powerful tool for studying and exploiting parallelism, specially for nested loop structures that offer the most fruitful source of parallelism in serial programs. But in general, loops whose statements are involved in a dependence cycle are considered to be serial. However in this paper we present how the framework of linear loop transformations can be extended in different ways. The framework presented solves the fundamental problem of deciding which transformation to apply. We focused this paper on the problem of extracting all the parallelism that may be present in a loop. The technique presented in this paper is applicable to codes whose full potential parallelism can not be exploited using linear transformations.

We have shown from a classical example taken from [11], which has been used by several authors [15,12], how our proposed method can be applied. For this example the authors referred present methods based on the technique named "Cycle Shrinking". Basically, they group the index points into packets. These packets are executed sequentially and all the index points within a packet are executed concurrently. They obtain a code with an outermost sequential loop that has about N/4 iterations. Our method presents important improvements: (a) The sequential loop has about N/9 iterations, as a consequence more iterations are executed in parallel. (b) In our method, the sequential loop becomes the innermost loop, this causes the code to be completely parallel. Another important effect is that the method can eliminate the barrier synchronization that is required using the Cycle Shrinking method.

In this particular case, the method proposed does not balance the load assigned to the processors if as many processors as parallel iterations are allocated. The number of tasks generated is greater than the parallelism evaluated [1] and tasks take different time to complete their execution. In [18] this scheme is extended to improve load balancing among processors, reducing the number of parallel tasks without increasing the execution time of the parallel loop.

Next we briefly comment some open questions left by the work presented in this paper. The partitioning method presented assumes the existence of a hamiltonian recurrence in the graph. This is not the common case, so the problem must be taken into consideration and heuristics to find a good solution proposed. Loop distribution can also be used when a hamiltonian recurrence is not present in the graph.

Acknowledgments

This work has been partially supported by the Ministry of Education of Spain under contracts TIC 880/92 and TIC429/95, Esprit BRA APPARC under grant no. 6634 and by the CEPBA (European Center for Parallelism of Barcelona).

References

[1] Ayguadé E., Labarta J., Torres J., Llabería J.M. and Valero M., Parallelism Evaluation and Partitioning of Nested Loops for Shared Memory Multiprocessors, chapter 11 of *Advances in Languages and Compilers for Parallel Processing*, The MIT Press, 1991.

[2] Ayguadé E. and Torres J., Partitioning the Statement per Iteration Space Using Non-singular Matrices, in *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.

[3] Banerjee U., *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[4] Banerjee U., Unimodular Transformations of Double Loops, chapter 10 of *Advances in Languages and Compilers for Parallel Processing*, The MIT Press, 1991.

[5] Banerjee U., *Loop Parallelization.* A book series on Loop Transformations for Restructuring Compilers Klumer Academic Publishers, Norwell, Massachussetts. 1994.

[6] Darte A., Risset T., Robert Y. Loop Nest Scheduling and Transformations. In J.J. Dongarra and B. Tourancheau, editors, Environment and Tools for Parallel Scientific Computing, volume 6 of *Advances in Parallel Computing*, North Holland, 1993.

[7] Fernández A., Llabería J.M. and Valero-García M., Loop Transformation using non-unimodular matrices, to be published in *IEEE Transactions on Parallel and Distributed Systems.*

[8] Knijnenburg P. M. W., Ayguadé E. and Torres J., Multitransformations: Code Generation and Validity. Technical Report 95-12, Dept. of Computer Science, Leiden University, 1995.

[9] Li W. and Pingali K., A Singular Loop Transformation Framework Based on Non-Singular Matrices, in *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computers*, New Heaven (CN), August 1992.

[10] Polychronopoulos C., *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[11] Polychronopoulos C. P., Compiler Optimization for Enhancing Parallelism and Their Impact on Architecture Design, *IEEE Transactrions on Computers* 37(8) (Aug. 1988) 991-1004.

[12] Robert Y. and Song S., Revisiting cycle shrinking, Parallel Computing 18 (1992), North-Holland, 481-496.

[13] Schrijver A., *Theory of Linear and Integer Programming*, John Wiley and Sons, 1986.

[14] Shang W. and Fortes J., Independendent Partitioning of Algorithms with Uniform Dependencies,

[15] Shang W. , O'Keefe M. and Fortes J. A., On Loop Transformations for Generalized Cycle Shrinking, *IEEE Transaction on Parallel and Distributed Systems*, Vol. 5, No. 2, February 1994, 193-204.

[16] Torres J., Ayguadé E., Labarta J. and Valero M., Align and Distribute-based Linear Loop Transformations, *Languages and Compilers for Parallel Processing*, 6th International Workshop, Portland Oregon, Lectures Notes in Computer Science 768, Springer-Verlag, 1993

[17] Torres J., Ayguadé E., Labarta J. and Valero M., Revisiting Framework of Linear Loop Transformations to Extract Full Loop Parallelism, Department of Computer Architecture, Polytechnic University of Catalunya, April 1995, CEPBA Research Report RR-95/09

[18] Torres J., Automatic Parallelization of Sequential Loops with Recurrences , Ph.D. Thesis, Department of Computer Architecture, Polytechnic University of Catalunya, November 1993, CEPBA Research Report RR-94/01

[19] Wolf M.E. and Lam M.S., A Loop Transformation Theory and an Algorithm to Maximize Parallelism, *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, October 1991.

[20] Wolfe M., *Optimizing Supercompilers for Supercomputers*, The MIT Press, 1989.

[21] Wolfe M., More Iteration Space Tiling, *Proceedings of the Supercomputing'89*, 1989.

[22] Xue J., Automating non-unimodular loop transformations for massive parallelism, Parallel Computing 20 (1994), North-Holland, 711-728.