Finding Free Schedules for Non-uniform Loops

V. Beletskyy, K. Siedlecki

Faculty of Computer Science, Technical University of Szczecin, Zolnierska 49 st.,
71-210 Szczecin, Poland, fax. (+4891) 487-64-39
vbeletskyy@wi.ps.pl, ksiedlecki@wi.ps.pl

Abstract. Algorithms, permitting us to build free schedules for perfectly and imperfectly nested affine loops with non–uniform dependences, are presented. The operations of each time schedule can be executed as soon as their operands are available. This allows us to extract maximum loop parallelism. The algorithms require exact dependence analysis. To describe and implement the algorithm and carry out experiences, the dependence analysis by Pugh and Wonnacott has been chosen where dependences are found in the form of tuple relations. The algorithms can be applied for both non-parametrized and parametrized loops. The algorithms proposed has been implemented and verified by means of the Omega project software.

1 Introduction

The larger number of transformations have been developed to expose parallelism in loops, minimize synchronization, and improve memory locality in the past, for example, [3]-[10], [14], [15]. Most of those transformations permit us to extract parallelism from both uniform and non-uniform loops.

A number of publications are devoted to loop parallelization with non-uniform dependences only [3], [4], [6], [7], [8], [10], [13], [14], [17], [20]. Techniques proposed are based on non-linear transformations, substituting non-uniform dependences with uniform ones, or applying linear programming techniques.

But the question is how much loop parallelism these approaches extract.

The general problem of the parallelism detection is the following. Dependences in loops can be represented in different ways, with approximations in general, or with an exact representation when possible. For each dependence representation based on approximations (level of dependences, uniform dependences, polyhedral representation of dependences), optimal algorithms exist, but for exact affine dependences, it is not known what are the loop transformations that extract maximal parallelism [5].

Following Vivien [19], we imply that an algorithm extracting parallelism is optimal if it finds all parallelism: 1) that can be extracted in its framework or 2) that is contained in the representation of the dependences it handles or 3) that is contained in the program to be parallelized (not taking into account the dependence representation used nor the transformations allowed).

Free schedules [5] permit us to extract all parallelism in loops, but most known techniques based on linear or affine schedules sometimes fail to find free schedules for non-uniform loops.

This paper presents two algorithms permitting us to find free schedules for perfectly and imperfectly nested loops with affine dependences. These algorithms are optimal and find all parallelism contained in an affine loop.

Our approach is based on non-linear schedules that can be implemented with procedures using a set of routines for manipulating linear constraints over integer variables, Presburger formulas, integer tuple relations and sets, and routines for generating code to scan the points in the union of a number of convex polihedra.

2 Background and definitions

In this section, we, in brief, attach well-known knowledge to explain better our algorithm for finding free schedules.

In this paper, we deal with affine loop nests where lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters, and the loop steps are known constants.

Following work [19], we refer to a particular execution of a statement for a certain iteration of the loops, which surround this statement, as an operation.

Two operations I and J are dependent if both access the same memory location and if at least one access is a write. We refer to I and J as the source and destination of the dependence, respectively, provided that I accesses the same memory location earlier than J.

Definition 1. 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 I in terms of the dependence destinations J (I= f(J)) or vice versa.

The algorithm considered in this paper requires an exact representation of the dependences, and hence exact dependence analysis.

Definition 2[18]. Dependence analysis is exact if for any affine dependence it detects a dependence if and only if one exists.

In general, any known technique, extracting exact dependences, can be applied to implement our algorithm. But the description of the algorithm and carrying out experiments depend on the format of the presentation of exact dependences.

To describe the algorithm and carry out experiences, we have chosen the dependence analysis proposed by Pugh and Wonnacott [18] where dependences are presented with dependence relations.

Definition 3[18]. 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.

Each dependence relation represents a dependence class (flow, anti, output).

The dependence analysis by Pugh and Wonnacott is implemented in Petit, a research tool for doing dependence analysis and program transformations.

Definition 4[5]. A schedule is a mapping which assigns a time of execution to each operation of the loop in such a way that all dependences are preserved, that is, mapping $\sigma: I \rightarrow Z$ such that for any arbitrary dependent operations op1 and op2 $\in$ I, $\sigma(op1) < \sigma(op2)$ if op1 $\prec$ op2.

Definition 5[5]. A free schedule assigns operations as soon as their operands are available, that is, mapping $\sigma: I \rightarrow Z$ such that

$$s(p) = \begin{cases} 0 \text{ if there is no } p' \in I \text{ s.t. } p' \prec p \\ 1 + \max(s(p'), p' \in I, p' \prec p) \end{cases} . \quad (1)$$

The free schedule is the "fastest" schedule possible. Its total execution time is

$$T_{free} = 1 + \max(s_{free}(p), p \in I) . \quad (2)$$

The algorithm considered in this paper is applicable for the loops that meet the requirements of the dependence analysis by Pugh and Wonnacott [18].

To follow the material of this paper, the reader should be familiar with the operations on tuple relations such as: union, difference, range, domain, application as well as existentially quantified variables explained in [11].

3 Free schedules for perfectly nested loops

A perfectly nested loop contains all statements within the innermost nest.

We will refer to the source of dependence as the fair dependence source if it is not a destination of any other dependence.

We define the length of a chain of synchronization between a pair of the dependent statement instances as N-1, where N is the maximal number of the statement instances that this chain connects.

The idea of the algorithm presented in this section is as follows. We divide all operations originated by each statement into two sets that contain the independent and dependent operations (sources and destinations), respectively. For the second set, we firstly find those operations for which all operands are available. They form the operations of layer0 to be executed firstly. Next, we eliminate from the second set the operations of layer0 and find again those operations for which all operands are available, etc. until there are no operations in the second set. The operations of the first set can be combined with the operations of arbitrary levels.

Algorithm 1. Find a free schedule for a perfectly nested loop

1. Find all loop carried dependences presented as tuple relations.
2. Build a relation R as the union of all the relations found in step 1.
3. Find the dependence sources I as the domain of the relation R.
4. Find the dependence destinations J as the range of the relation R.
5. Find independent operations IND, that is, those operations that do not belong to any pair of dependent operations as follows IND:=IS-I-J, where IS is the set of operations belonging to the loop iteration space.

6. Find all fair dependence sources FS as the difference between J and I, that is, FS:=I-J. They form the operations of layer 0 to be executed firstly.
7. Find the set of the dependence destinations L_1 whose sources belong to the set FS by means of the following application operation L_1:=R(FS), that is, relation R is applied to set FS. Note, that the set L1 contains the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length one or more.
8. Find the operations, belonging to the first layer, as follows: Lay_1:=L_1-R(J). The dependence destinations, belonging to R(J), are linked with the fair dependence sources by a chain of synchronization of length two or more. Hence, the set of the dependence destinations L_1-R(J) contains only those dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length one.
9. Find the second and remaining layers of the dependence destinations as follows:

   i=2;
   Loop:

$J:=J-Lay_{i-1}$	- eliminating the dependence destinations belonging to layer i-1
$L_i:=R(Lay_{i-1})$	- finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length i or more
$D_i:=R(J)$	- finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length i+1 or more
$Lay_i:=L_i-D_i$	- finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length i

   if Lay_i ==False then the end; else i=i+1; goto Loop.

The independent operations IND can be united with arbitrary layers.
Let us illustrate the presented algorithm by means of the following loop

```
1:real a(50,50)
2:  for i=1 to 6 by 1 do
3:    for j=1 to 10 by 1 do
4:       a(i+j,3*i+j+3)=a(i+j+1,i+2*j+4)
5:   endfor
6:  endfor
```

1. The loop above exposes the following dependences found with Petit (Fig. 1)

   (anti dep.) R1:= $\{[i,2i] \rightarrow [i,2i+1]: 1 \le i \le 4\}$

   (anti dep.) R2:= $\{[i,j] \rightarrow [i',i+I'+1]: j=2i' \;\&\&\; 1 \le i < i' \le 5\}$

   (flow dep.) R3:= $\{[i,j] \rightarrow [j-i-1,2i]: 2i+2 \le j \le i+7,10 \;\&\&\; 1 \le i\}$

2. The relation which unites all the dependences above is of the form

R:= R1 union R2 union R3

3. All the dependence sources I in the loop iteration space are found as follows

   I:= domain R

4. All the dependence destinations J in the loop iteration space are

   J:= range R

5. The independent operations IND are represented as the following set

   IND:=IS-I-J, where
   IS:={[i,j]: 1≤i≤6 && 1≤j≤10}

6. All the fair dependence sources FS are represented with the set (Fig. 2)

   FS:=I-J

7. The set of the dependence destinations L1 whose dependence sources belong to the set FS are

   L1:= R(FS)

8. The operations, belonging to the first layer, are as follows (Fig. 3)

   Lay1:= L1-R(J)

9. The second layer of the dependence destinations is found as below (Fig 4)

   J:= J – Lay1 = {[i,2i+1]: 2 <= i <= 4}
   L2 := R(Lay1) = {[i,2i+1]: 2 <= i <= 4}
   Lay2 := L2 - R(J) = {[i,2i+1]: 2 <= i <= 4}

The third layer is empty. This means that there are no operations in this and further layers. The independent operations IND can be united with arbitrary layers. In our implementation, we unite them with fair dependence sources to build layer 0.

On the basis of sets, representing layers, code can be generated by means of standard techniques and tools, for example, with the Omega calculator or Polylib code generator. All the same instances of all statements in a perfectly nested loop belong to the same layer.

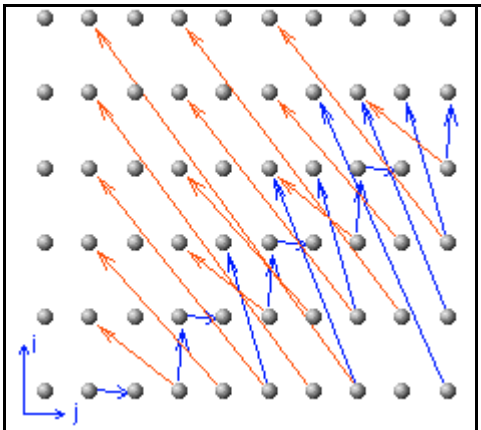

Figure 1. Iteration space with the dependency vectors

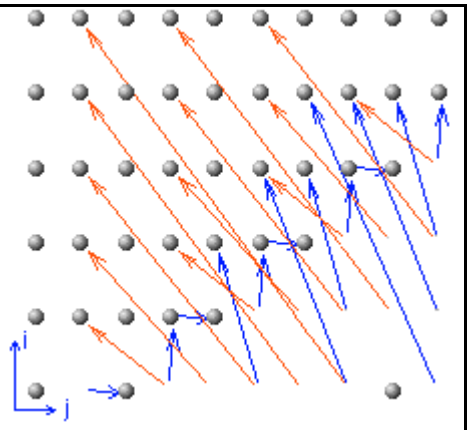

Figure 2. Iterations space without the fair sources

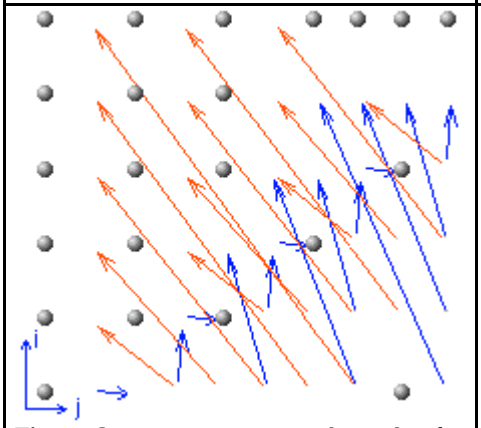

Figure 3. Iteration space without the fair sources and operations belonging to the first layer

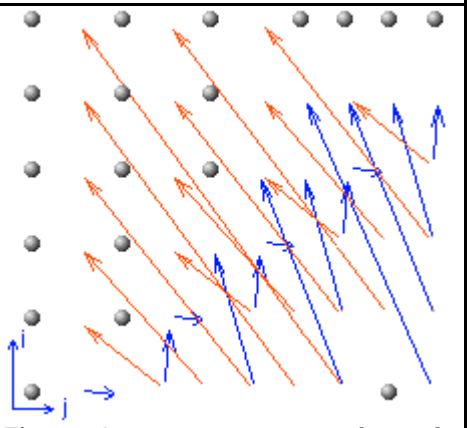

Figure 4. Iteration space without the dependent operations (all layers were eliminated)

4 Free schedules for imperfectly nested loops

Algorithm 1 deals with the loop statements as one indivisible block including all statements since in perfectly nested loops, the domains of statements are the same.

In imperfectly nested loops, statements may have different domains, and, in general, we should deal with each statement independently. To generate resulting code, we should find free schedules for each statement independently, and next combine the same layers (belonging to the same schedule time) originated by different statements into one resulting layer.

Algorithm 1. Find a free schedule for an arbitrary nested loop

1. Find all cross-iteration dependences as well as the dependences originated by statements within the loop body. Build two sets of the dependences for each statement j, j=1,2,...,n. The first one, $S1_j$ includes the dependence relations whose destinations are the instances of statement j. Let the relations of $S1_j$ be $R1_{kj}$, $k_j=1,2,...,n_j$, n_j is the number of the dependence relations in $S1_j$. The second set, $S2_j$ includes the dependence relations whose sources are originated with statement j.

For each statement j do:

2. Find the sources of the dependences as the domains of the relations, belonging to set $S2_j$, and unite them into one set I_j.
3. Find the destinations of the dependences as the ranges of the relations, belonging to set $S1_j$, and unite them into one set J_j.
4. Find the independent operations IND_j, that is, those that do not belong to any pair of the dependences as follows $IND_j:=IS_j-I_j-J_j$, where IS_j is the set representing the iteration space of statement j.
5. Find all fair dependence sources FS_j as the difference between I_j and J_j. Sets FS_j form layer 0 of the operations which should be executed firstly.
6. Find the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length one or more as follows $L1_{kj}:=R1_{kj}(FS(R1_{kj}))$, $k_j=1,2,...,n_j$, where $R1_{kj}$ are found in step 1, $FS(R1_{kj})$ are the sets of the fair dependence sources that are the sources of the dependences represented with relations $R1_{kj}$. Unite sets $L1_{kj}$ into one set $L1_j$.
7. Find the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length two or more as follows $D1_{kj}:=R1_{kj}(J(R1_{kj}))$, $k_j=1,2,...,n_j$, where $J(R1_{kj})$ are the sets of the dependence destinations that are the dependence sources represented with relations $R1_{kj}$. Unite sets $D1_{kj}$ into one set $D1_j$.
8. Find the operations belonging to the first layer as $Lay1_j:=L1_j-D1_j$.

End for each

9. Find the second and remaining layers of the dependence destinations as follows:

i=2;
For each j
Loop:

$Jj:=Jj-Lay(i-1)_j$;	-elimination of the dependence destinations belonging to layer i-1 and originated by the instances of statement j
$Li(R1_{kj}):=R1_{kj}(Layi(R1_{kj}))$, $k_j=1,2,...,n_j$;	- finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length i or more, $Layi(R1_{ki})$ are the sets representing the operations of layer i and originated by the sources of the dependences represented with relations $R1_{kj}$
unite sets $Li(R1_{kj})$ into one set L_{ij};	
$Di_{kj}:=R1_{kj}(J(R1_{kj}))$, $k_j=1,2,...,n_j$;	- finding the dependence destinations that are linked with the fair dependence

sources by a chain of synchronization of length i+1 or more

unite sets Di_{kj} into one set D_{ij};
Lay_{ij}:=L_{ij}-D_{ij};

- finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length i

End for each
if each Lay_{ij} ==False then the end; else i=i+1; goto Loop;

The independent operations INDj can be combined with arbitrary layers. In our implementation, we unite them with the fair dependence sources to build layer0.

Consider the following loop.

```
1:  for i=1 to 100 by 1 do
2:     a(2*i+4,i+5)=c(i,i)
3:     for j=1 to 100 by 1 do
4:        b(i,j)=a(i+2*j+2,i+j)
5:     endfor
6:  endfor
```

1. It exposes the following dependences found with Petit

   (flow dep. 2→4) R1 := {[8]→[8,5]}

   (flow dep. 2→4) R2 := {[i]→[8,i-3]: 4 ≤ i ≤ 7 }

   (anti dep. 4→2) R3 := {[8,j]→[j+3]: 6 ≤ j ≤ 97 }.

   For statement 2, we form the following sets of the relations:

   S12:={R3}, S22:={R1,R2}.

   For statement 4, we construct:

   S14:={R1,R2}, S24:={R3}.

2. For statement 2, we first unite relations R1 and R2 because they describe the dependences between the same statements (2 → 4) and then find the following dependence sources

   R2_24:= R1 union R2, I2:= domain R2_24.

   For statement 4, the set of the dependence sources is

   I4:= domain R3.

3. For statement 2, we find the following dependence destinations

   J2:=range R3.

For statement 4, we first unite relations R1 and R2, and then find the dependence destinations as follows

R1_24:= R1 union R2, J4:= range R1_24.

4. The fair dependence sources are found by means of the following operations on sets:

   FS2:= I2-J2 = {[8]} union {[i]: $4 \le i \le 7$.}

   FS4:= I4-J4 = {[8,j]: $6 \le j \le 97$}.

5. We find the sets of operations L12, L14, containing the dependence destinations that are linked with the fair dependence sources FS4 and FS2, respectively, by a chain of synchronization of length one or more, by means of the application operations

   L12:= R3(FS4), L14:=(R1 union R2)(FS2).

6. The dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length two or more are as follows

   D12:= R3(J4) = {[In_1]: FALSE }

   D14:= (R1 union R2)(J2) = {[i,j]: FALSE }.

7. The operations, belonging to the first layer, are as below

   Lay12:= L12 - D12 = {[i]: $9 \le i \le 100$}

   Lay14:= L14-D14 = {{[8,j]: $1 \le j \le 5$}.

8. To find the dependence destinations, belonging to the second layer, we first find

   J2:= J2-Lay12 = {[i]: FALSE }

   J4:= J4-Lay14 = {[i,j]: FALSE }.

   This means that there are no dependence destinations in the second layer.

9. Finally, we find the independent operations as follows

   IS2:={[i]: 1<=i<=100}, IND2:=IS2-I2-J2,

   IS4:={[i,j]: 1<=i,j<=100}, IND4:=IS4-I4-J4,

which can be combined with arbitrary layers.

5 Parameterized loops

The algorithms presented permits us to find free schedules not only for the loops with the known before compilation loop bounds but also for parameterized loops.

Following the algorithm presented, we can find the given number of parametrized layers for a loop under the free schedule. If for a certain layer, there are no operations, this means that the loop is characterized by the constant number of layers and each of them can be presented in a parametrized form. The procedure described permits us to reveal whether the number of layers is irrelevant to the size of a loop. If such a loop is detected, code may be generated with a complexity that does not depend on the loop size.

But the main problem that requires further research is how to find the number of layers for parameterized loops in the general case.

If the number of layers does not depend on the size of a loop, we can present the free schedule in a symbolic way and to generate corresponding code with a complexity that depends on the loop but not on the volume of computation it describes.

When there exist only two layers, this means that there are no operations that simultaneously are the sources and destinations of dependences. Such a case can be very easily reviled. We should form the set of the dependence sources and the set of the dependence destinations originated with all loop statements and next find the intersection of these sets. If the result is FALSE, this means that there exist only two layers under the free schedule.

Consider the following loop

```
for(i=1;i<=n;i++) {
  a[2*i+4][i+5]=b[i][i];
  for(j=1;j<=n;j++)
    c[i][j]=a[i+2*j][i+j];
}
```

Carrying out the following parametrized calculations

$I2 := \{[i]: 2 \le i \le 6 \;\&\&\; 6 \le n\}$

$I4 := \{[6,j]: 6 \le j < n\}$

$J2 := \{[i]: 7 \le i \le n\}$

$J4 := \{[6,j]: 1 \le j \le 5 \;\&\&\; 6 \le n\}$

$I2\_J2 := I2 \wedge J2 = \{[i]: FALSE\}$

$I4\_J4 := I4 \wedge J4 = \{[i,j]: FALSE\}$,

we can conclude that there are no common operations in sets I2, J2 and in sets I4, J4, so there exist only two layers under the free schedule for the loop considered.

Table 3 shows the parametrized code for the loop above.

Table 1. The left and right-hand sides of Table 1 present the code for layer 0 and layer 1, respecrively

If (n ≥ 6) { par for(i=2;i≤6;i++) {	if (n ≥ 6) { par for(j=1;j≤5;j++)

```
if (i ≥ 6) {
 par for(j=1;j≤5;j++)
  c[6][j]=a[6+2*j][6+j];
}
a[2*i+4][i+5]=b[i][i];
}
```

```
 c[6][j]=a[6+2*j][6+j];
}
par for(i=7;i≤n;i++)
 a[2*i+4][i+5]=b[i][i];
```

It is worth to note that in the case when a loop exposes the constant number of layers, free schedules may be much faster than the best affine schedules. Consider the following example [8]

```
for (i=0; i≤2*n; i++)
 a[i]=a[2*n-i];
```

The best affine schedule for this loop is i/2 [8], that is, the number of the layers yielded with this schedule equals to n.

Algorithm 1, presented in Section 3, finds only two layers for this loop. The code is as follows

```
par for (i=0;  i≤n-1; i++){
 a[i]=a[2*n-i];
 a[2*n-i]=a[i];
}
if (n ≥ 1)
 a[n] = a[2*n-n];
```

The first and second statements in the loop originate the operations of layer0 and layer1, respectively. The condition statement represents the independent operations.

6 Experiments

The goals of the experiments were to verify the correctness of the algorithms presented, study how the time of the generation of layers and code depend on the loop bounds, the number of dependence relations, and the number of layers generated for several working non-parameterized loops as well as to evaluate the speedup of the code generated.

The algorithms, presented in Sections 3 and 4, were implemented by means of the Omega project software [ftp://ftp.cs.umd.edu/pub/omega]. After finding free schedules for a loop, code was generated by the Omega codegen and then translated into C sources.

Experiments were carried out on the PC computer: Processor - Athlon 1600+ (≈1400 MHz), RAM -256 MB, OS -Windows 2000.

The correctness of the code generated was verified by means of the comparison of the results obtained by the execution of an original and transformed loops.

Tables 1 and 2 present source loops, their bounds, the number of dependence relations found with Petit, the number of layers, the layers and code generation times, and the theoretical speedup (only for perfectly nested loops) for perfectly and imperfectly nested loops, respectively.

As follows from the results presented in these tables, the layers and code generation times depend on the loop bounds, the number of dependence relations, and the number of layers generated. As the number of dependence relations grows, relations for generating layers become more complicated that increases the layers generation time. The influence of the loop bounds and the number of layers on the layers and code generation times is obvious. The number of layers defines minimal synchronization time.

The theoretical speedup of parallel code, supposing the unlimited number of processors, can be presented as follows:

$$S_{theoretical} \leq 1/\sigma,$$

where σ - fraction of the computation to be executed serially.

The formula above is based on Amdahl's low, which states that speedup is limited by the time taken to do the serial calculations of the application. In our case, for perfectly nested loops, σ is defined as the ratio of the number of layers (defines the time of sequential execution) to the number of all iterations. The theoretical speedup gives maximally possible speedup. In practice, it is very difficult to reach the theoretical speedup since the formula above does not take into account such important properties of a parallel program as locality, synchronization, and the volume of communication. But the theoretical speedup can be used for evaluating the quality of parallel programs under development as well as for assessing maximal parallelism.

To find the speedup of the codes generated, we have parallelized them by the Omni OpenMP compiler. Each parallel program executes all iterations of each layer in parallel, while, in the end of each layer, an instruction of barrier synchronization is introduced. We have executed parallel programs on computers with four processors (Windows NT) and with two processors (Linux). For each parallel code generated for the loops presented in Tables 1 and 2, we became positive speedup.

Figure 1 shows how the speedup of the parallel code, generated for first loop (Table1), depends on the average time of the execution of one loop iteration. At multithreading, speedup greatly depends on the time of the thread execution that in turn depends on the average run time of one loop iteration.

Table 2. Experiments with perfectly nested loops

Nr	Loops	Loop bounds M=N=…	Dep. relations	layers	Dep. iterations, %	Layers generation time (s)	Code generation time (s)	Theoretical speedup

1	for(i=1;i<M;i++)
	for(j=1;j<N;j++)
	a[2*j+3][i+j+3]=
	=a[2*i+j-1][3*i-1];
		100	2	5	22	0,04	0,17	1960

		500		6	22	0,04	0,25	41500

2	for(i=1; i<N; i++)
	for(j=1; j<M; j++){
	a[3*i][4*j]=a[i][j];
	a[2*i-][3*j]=a[i][j];
	}
		50	9	6	29	0,761	9,57	400

3	for(i=1;i<N;i++)
	for(j=1;j<M;j++)
	a[2*i-1][2*j-1] = a[i][j];
		100	3	7	25	0,04	0,401	1400

		500		9	25	0,06	0,51	27666
		1000		10	25	0,071	0,561	998001

4	for(i=1;i<N;i++)
	for(j=1;j<M; j++)
	a[2*j+3][i+1]=
	=a[2*i+j+1][i+j+3];
		10	1	2	11	0,001	0,01	40,5

		100		5	23	0,01	0,05	1960
		500		7	24	0,04	0,13	35571
		1000		8	24	0,001	0,09	124750

5	for(i=0;i<N;i++)
	for(j=0;j<M;j++)
	a[i+2*j][3*i+j+3]=
	=a[i+2*j+1][i+2*j+4];
		100	2	3	14	0,01	0,07	3333

		500			13	0,01	0,06	83333
		1000			13	0,01	0,06	333333

6	for(i=0;i<N;i++)
	for(j=0;j<M;j++)
	a[i+j+50][3*i] = a[i][j];
		100	1	2	11	0,001	0,01	5000

		500		4	24	0,01	0,05	62500

7	for(i=0; i<N; i++)
	for(j=0; j<M; j++)
	a[19*i+3][j] =
	a[2*i+21][j];
		100	1	3	5	0,001	0,01	3333

		500			5	0,001	0,02	83333
		1000		4	5	0,01	0,02	250000

8	for(i=0; i< N; i++)
	for(j=0; j< M; j++) {
	a[11*i][5*j] =a[i][j];
	a[10*i+1][j] =a[i][j];
	}
		100	12	5	13	0,11	0,391	2000

		500		7	13	1,151	4,146	35714
		1000		8	13	1,212	4,867	125000

9	for(i=1;i<M;i++) {
	a[10 * i]=a[i];
	a[11 * i]=a[i];
	a[12 * i]=a[i]; }
		100	18	6	30	0,03	0,08	16,5

		200		7	30	0,05	0,21	28

Table 3. Experiments with imperfectly nested loops

Nr	Loops	Loop bounds M=N=O=…	Dep. relations	layers	Dep. iterations %	Layers generation time (s)	Code generation time (s)

1	for(i=1;i<=M;i++) {
	a[2*i+4][i+5]=b[i][i];
	for(j=1;j<=N;j++)
	c[i][j]=a[i+2*j][i+j];
	}
		10	3	2	16	0	0,08

		50		2	3,8	0,01	0,11
		100		2	2	0,02	0,07

2	for(i=0;i<=2*M;i++) {
	a[i][0]=a[2*100-i][100];
	for(j=1;j<=N;j++)
	a[i][j]=a[i][j-1];
	}
		10	2	11	100	0,01	0,03

		50	2	51	100	0,04	0,13
		100	5	202	100	9,784	0,43

3	for(i=1;i<=M;i++) {
	for(j=1;j<=i;j++) {
	a[i-j][i+j]=b[i][i];
	for(k=j;k<=i;k++)
	c[i][i]=a[i+2*k+5][4*k-j];
	}
	}
		10	2	55	80	0,28	0,571

		20		210	87	3,855	5,518
		30	3	465	91	25,606	29,092

4	for(k=1;k<=M;k++) {
	a[k][k]=a[k][k];
	for(i=k+1;i<=N;i++) {
	a[i][k]=a[i][k]*a[k][k];
	for(j=k+1;j<=i;j++)
	a[i][j]=a[i][j]-a[i][k]*a[j][k];
	}
	}
		10	19	28	100	0,08	0,09

		50		148	100	0,38	0,431
		100		298	100	0,751	0,871

5	for(i=1;i<=M;i++) {
	for(j=1;j<=N;j++) {
	a[i-j+100][i]=b[i][j];
	for(k=1;k<=O;k++)
	c[i][j]=a[100-j][k];
	}
	}
		10	4	28	95	0,18	1,863

		20		58	97,5	0,741	11,857
		30		88	98,(3)	1,972	40,358

6	for(k=1;k<=M;k++) {
	for(i=k+1;i<=N;i++) {
	a[i][k]=a[i][k]*a[k][k];
	for(j=k+1;j<=i;j++)
	a[i][j]= a[i][j]-a[k][j]*a[i][k];
	}}
		10	12	18	100	0,03	0,06

		50		98	100	0,17	0,28
		100		198	100	0,31	0,571

Creating and synchronizing threads need certain time. If this time is comparable or more than the time of the thread execution, the efficiency of parallel applications may be low enough. This is why we have investigated how speedup depends on the average run time of one iteration.

It is worth to note that there are possibilities to improve the codes generated. Since there may exist independent iterations in loops, we may combine any of them with an arbitrary layer of dependent iterations to improve the regularity of code generated.

We have observed that in most loops investigated, the number of iterations decreases with each next layer (but in general this is not a rule). Sometimes, it can be reasonable to combine several last layers into one thread to diminish the synchronization overhead. Currently, we are investigating these possibilities.

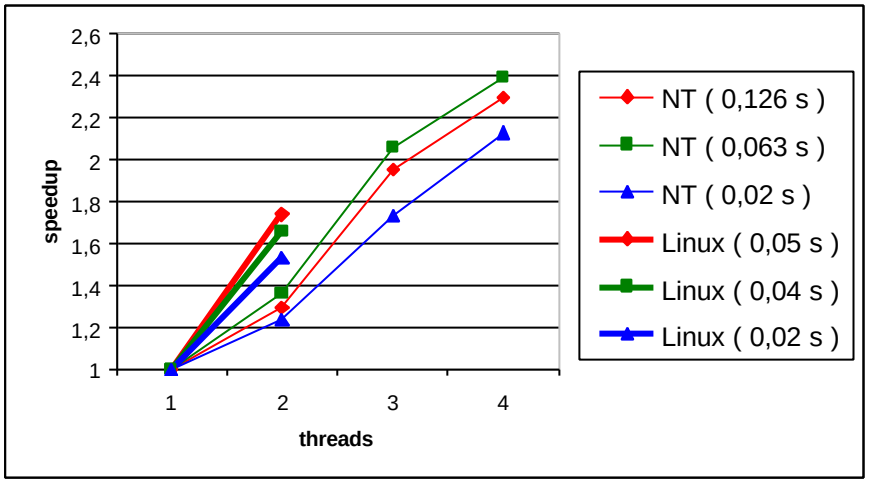

Fig. 5. The influence of an average run time of one iteration on speedup for first loop from Table1

7 Related work

The approaches presented in [1], [13], [16], [17] build an explicit graph of a subset of the iteration space, with each node representing the instance of a statement. Free schedules can be found by searching the graph or using the transitive closure of the graph, but dependences are restricted to uniform ones and the problem regarding boundary cases exists. We use tuple relations as an abstraction for data dependences. This permits us to reach all the same merits that discussed in [12]: handle non-uniform dependences and multidimensional loops without having to make special checks in boundary conditions. But, in contrast to work [12], our technique does not require the calculation of the transitive closure of the dependence relations. The exact transitive closure of an affine integer tuple relation may not be affine that limits the use of the technique presented in [12].

Concerning parallelism detection, the following facts are known.

If the level of dependences is the only available representation, then Allen and Kennedy's algorithm (loop distribution) is known to be optimal [2] with respect to its

input, that is, it finds maximal parallelism that is contained in the representation of the dependences it handles.

For the case of a single statement with uniform dependences, linear scheduling (optimized Lamport's hyperplane method) is asymptotically optimal [4].

For the case of several statements with uniform dependences, the previous result has been extended in work [9] to show that a linear schedule plus shifts leads to finding maximum parallelism.

For the case of polyhedral approximations of dependences (including direction vectors), unimodular transformations plus shifts plus distribution (Darte and Vivien's algorithm) is optimal [3].

For affine dependences, the most powerful algorithms is Feautrier's algorithm based on multi-dimensional affine schedules [8]. But as mentioned by Feautrier, it is not optimal for all codes with affine dependences. However, among all possible affine schedules, it is optimal [19].

The approaches devoted only loops with non-uniform dependences […] extract the different amount of loop parallelism, but none from them extract maximal parallelism.

So, one of the remaining questions is the following. For affine dependences, can we keep some regularity and achieve maximal parallelism? What type of transformations (or code rewriting) should be done to get optimal parallelism? The only attempt in this direction is index splitting [8]. But it is a heuristic procedure, and it is not known how much index splitting is necessary, i.e., how many different code structures must be generated to reach optimality.

The technique presented in this paper is a first step to understand the problem of free schedules of loops with affine dependences. It permits us to find free schedules for both non-paramerized and parametized loops but does not answer what in general is the number of layers (times) under the free schedule for a parameterized loop.

8 Conclusion

In this paper, we have presented the algorithm that permits us to build free schedules for both non-parametrized and parametrized loops. Code can be produced easily by means of well-known techniques and public available tools, for example, the Omega project software (ftp://ftp.cs.umd.edu/pub/omega) or the Polylib code generator (http://www.irisa.fr/cosi/ALPHA/welcome.html).

The algorithm proposed has been implemented and verified by means of the Omega project software.

It is worth to note that focusing on the free schedule is very important for understanding the parallelism contained in a set of loops, not necessarily for its run-time execution.

The tasks for further research are as follows. The number of steps of the algorithm must be determined for the general case of parameterized loops. The challenge here is to be able to compute a schedule in a symbolic way and to generate the corresponding code with a complexity that depends on the program but not on the volume of the computation it describes. The main theoretical question is: it is possible, for the case of affine dependences, to generate a compact code (i.e., with a

size that depends only on the initial program size), that leads to theoretical "performance" that is roughly as good as good as the free schedule? This challenge deserves more attention from the research community.

References

[1] Allen, R, Kennedy, K.: Optimizing Compilers for Modern Architectures, Morgan Kaufmann, 2001

[2] Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 39-50, April 1991.

[3] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. Computing Surveys, 26(4):345-420, December 1994.

[4] U. Banerjee. Loop Transformations for Restructuring Compilers. Kluwer Academic, 1993.

[5] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop paralellelization algorithms: from parallelism extraction to code generation. Technical Report 97-17, LIP, ENS-Lyon, France, June , 1997.

[6] Chen, D.-K.: Compiler optimizations for parallel loops with fine-grained synchronization. Technical report TR-1863, Department of Computer Science, University of Illinois at Urbana-Champaign (1994)

[7] Z. Chen and W. Shang, ``On uniformization of affine dependence algorithms," in Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, pp. 128-137, December 1992.

[8] J.-F. Collard, P. Feautreer, and T. Risset. Construction of DO loops from systems of affine constraints. Technical Report 93-15, LIP, Lyon, 1993

[9] Darte, A., Vivien, F.: On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Journal of Parallel Algorithms and Applications, Special issue on "Optimizing Compilers for Parallel Languages", Vol. 12 (1997) 83-112

[10] Darte, A., Vivien, F.: Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs, International Journal of Parallel Programming, Vol. 25(6) (1997) 447-496

[11] Darte, A., Khachiyan, L., Robert, Y.: Linear Scheduling is Nearly Optimal, Parallel Processing Letters, Vol. 1(2) (1991) 73-81

[12] Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhäuser Boston (2000)

[13] P. Feautrier. Dataflow analysis for array and scalar references. Int. J. of Parallel Programming, 20(1):23-53, feb.1991

[14] Feautrier, P.: Some efficient solutions to the affine scheduling problem, Part I, One-dimensional Time. Int. J. of Parallel Programming, Vol. 21(5) (October 1992) 313-348

[15] Feautrier, P.: Some efficient solutions to the affne scheduling problem, Part II, multidimensional time. Int. J. of Parallel Programming, Vol. 21(6) (December 1992)

[16] Feautrier, P., Griebl, M., Lengauer, C.: Index Set Splitting. International Journal of Parallel Programming, Vol. 28(6) (2000) 607-631

[17] Patrick Le Gouëslier d'Argence,: Affine Scheduling on Bounded Convex Polyhedric Domains is Asymptotically Optimal. TCS 196(1-2) (1998) 395-415

[18] J. Ju and V. Chaudhary, ``Unique sets oriented partitioning of nested loops with non-uniform dependences," in Proceedings of the International Conference on Parallel Processing, pp. III45--III52, 1996.

[19] Kelly, W., Pugh, W.: A framework for unifying reordering transformations. Technical Report CS-TR-2995.1, University of Maryland (April 1993)

[20] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega Library Interface Guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park (March 1995)

[21] Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive Closure of Infinite Graphs and its Applications. International Journal of Parallel Programming, Vol. 24(6) (December 1996) 579-598

[22] Kodukula, K. Pingali. Transformatations for Imperfectly Nested Loops In Proc. Supercomputing 96, Novemb

[23] Krothapalli, V.P., Sadayappan, P.: Removal of Redundant Dependences in DOACROSS Loops with Constant Dependences. In Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (July 1991) 51-60

[24] Lim, W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, Rhodes, Greece (June 1999)

[25] 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 (January 1997)

[26] Midkiff, S.P., Padua, D.A.: A comparison of four synchronization optimization techniques. In Proc. 1991 IEEE International Conf. on Parallel Processing (August 1991) II-9 – II-6

[27] Midkiff, S.P., Padua, D.A.: Compiler algorithm for synchronization. IEEE Trans. on Computers, Vol. 36(12) (1987) 1485-1495

[28] Pugh, W., Wonnacott, D.: An Exact Method for Analysis of Value-based Array Data Dependences. Workshop on Languages and Compilers for Parallel Computing (1993)

[29] S. Punyamurtula, V. Chaudhary, J. Ju, and S. Roy, “Compile time partitioning of nested loop iteration spaces with non-uniform dependences,”' Journal of Parallel Algorithms and Applications (special issue on Optimizing Compilers for Parallel Languages), October 1996.

[30] Quilleré, S. Rajopadhye, D. Wilde. Generation of Efficient Nested Loops from Polyhedra. International Journal of Parallel Programming, 28(5), Octobre 2000.

[31] T. H. Tzen, L. M. Ni,” Dependence uniformization A loop parallelization technique” IEEE transactions on Parallel and Distributed Systems, vol. 4, pp. 547--558, May 1993.

[32] Vivien F. On the optimality of Feautrier's scheduling algorithm. In Proceedings of the EUROPAR'2002 (2002)

[33] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. Transactions on Parallel and Distributed Systems, 2(4):452-470, October 1991.

[34] M. Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company,1995.

[35] J. Xue. Unimodular transformations of Non-Perfectly nested Loops *Parallel Computing*, 22(12):1621-1645, 1997.

[36] Xue, J.:Loop Tiling for Parallelism, Kluwer Academic Publishers, Boston Hardbound, 2000

[37] Zaafrani and M. Ito, ``Parallel region execution of loops with irregular dependences,'' in Proceedings of the International Conference on Parallel Processing, pp. II--11 to II--19, 1994.

[38] H. Zima and B. Chapman. Supercompilers for and Vector Computers. ACM Press, 1990.