N-Dimensional Perfect Pipelining

Ki-Chang Kim
Alexandru Nicolau

Technical Report No. 92-18

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

Hawaii Conf.

N-Dimensional Perfect Pipelining

Ki-Chang Kim and Alexandru Nicolau
Computer Science Department
University of California - Irvine
Irvine, CA 927171
Fax: (714)856-4056
E-mail: kkim@ics.uci.edu,nicolau@ics.uci.edu

Abstract

In this paper, we introduce a technique to parallelize nested loops at the fine grain level. It is a generalization of Perfect Pipelining which was developed to parallelize a single-nested loop at the fine grain level. Previous techniques that can parallelize nested loops , e.g. DOACROSS or Wavefront method, mostly belong to the coarse grain approach. We explain our method, contrast it with the coarse grain techniques, and show the benefits of parallelizing nested loops at the fine grain level.

1 Introduction

Loops are some of the richest program constructs where parallelism is available. Especially as the nest depth of the loop increases, the time that the CPU spends in it sharply climbs up. Many vectorization techniques have been developed to exploit the parallelism hidden in this construct [Kenn80][KKPL81][Wolf82][AlKe87]. For loops which are not vectorizable, however, the general technique is the Wavefront method [Mura71][Lamp74][Kuhn80].

An elegant way of implementing the Wavefront method through the combination of loop skewing and loop interchange is shown in [Wolf82]. [IrTr88] [Bane90] [LaWo90] show recent developments in this direction (see Section 3 for more detail). But since in the Wavefront method the unit of scheduling is an iteration, the parallelism *inside* iterations is not utilized. Each iteration is regarded as an atomic computational unit and executed by a single processor sequentially. This approach is useful to reduce the parallelizing complexity for nested loops, but it also reduces the amount of parallelism exploitable. Of course, the fine grain loop body parallelism may trivially be exposed *after* Wavefront method is applied, but this may miss some of the parallelism as we will see in section 4.1. Integrating fine and coarse grain parallelism is particularly important in light of the growing popularity of superscalar and VLIW machines such as the i860, i960, or the IBM R6000. In this context, machines such as the Touch Stone project make the exploitation of parallelism at all levels critical.

Parallelizing loops at the fine grain level has been pursued by other numerous researchers [Fish79] [Nico85] [GrLa86] [CCK87] [Lam87] [AiNi88]. For one dimensional loops, given enough resources, there exists an optimal solution [AiNi88]. Figure 1(a)-(b) shows an example loop and its parallelized form at the statement level. The parallelized form can be obtained by the following process. We unwind the loop repeatedly while scheduling each statement instance at the earliest cycle it can be executed (ASAP schedule) until a pattern is detected in the schedule. [1] Figure 1(c) shows this scheduling process, where the pattern

[1]Throughout this paper, "schedule" means a static reordering of the statements by the compiler. This

is enclosed with a box. Then, we replace the original loop body with this pattern. The schedule obtained this way is known to be optimal [AiNi88]. However, previous attempts to expose fine grain parallelism in nested loops have not been totally satisfactory. For example, Loop Quantization [Nico87] computes the amount of unwinding for nested loops, but does not maximize parallelism.

In this paper, we extend the approach in [AiNi88] to the n dimensional case to parallelize nested loops at the fine grain level. Finding an optimal schedule for the n dimensional case is an open problem. For some cases, we can easily find optimal schedules. Figure 2(a) shows such an example. In the figure, the dependence edges are annotated with the *dependence distance vectors*[2]. Thus, $(0,0)$ means an *in-loop* (non loop carried) dependence, $(0,1)$ means the dependence carried by the i_2 loop, etc. The ASAP schedule of this loop after some number of 2-dimensional unwindings is given in Figure 2(b). We observe that the delay of statements A and B along the i_1 dimension is always 2, while along the i_2 dimension it is always 1. For example, $A(i_1,i_2)$ can be executed only 1 cycle after $A(i_1,i_2-1)$ and 2 cycles after $A(i_1-1,i_2)$. A similar argument applies for statement B. Because of this regularity, we can parallelize the original loop as in Figure 2(c).[3] The parallelized loop is exposing *all* statement level parallelism in this loop. The reader can verify this by following its execution several steps. At each step, the loop correctly executes all statments that can be done as soon as possible.

However, in general, when we schedule all the statement instances ASAP, we do not necessarily see a fixed delay pattern emerge as in the previous example. One example is given in Figure 3. The delay pattern is given in Figure 3(c). In the figure, $d_1(i_1,i_2)$ is the delay along the first dimension at iteration (i_1,i_2), and $d_2(i_1,i_2)$ the delay along the second dimension at iteration (i_1,i_2). For example, the delay between A_{i_1,i_2} and A_{i_1+1,i_2} is represented by $d_1(i_1,i_2)$. Similarly, the delay between A_{i_1,i_2} and A_{i_1,i_2+1} is represented by $d_2(i_1,i_2)$. For both dimensions, the delays are not constant; they could be 1 or 2 depending on the values of i_1 and i_2. It is an open problem whether we can optimally parallelize loops whose delays are not constants but functions of index variables, as in this example. Furthermore, as will be shown in Section 4, an optimal solution would require knowing the exact bound of all the loops at compile time; since this information is not usually available, an optimal solution is, in general, only of theoretical interest.

Instead of trying to parallelize loops optimally for all cases of delay patterns, we simply force the delays to be constant, and compute the parallel form based on these delays.

2 Definitions

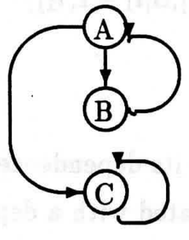
Before going into the details of our method, we need to define a few terms. We will use a tree, called *loop tree*, to capture the structure of a nested loop. In this tree, each node corresponds to a loop in the nested loop, except the leaf nodes which correspond to the statements. Since a statement is regarded as a loop with a single iteration, *loop* and *node* will be used interchangeably.

schedule is then executed *as is* , i.e. with the order of the statements in the schedule being preserved. This corresponds to the VLIW/superscalar model, and can be explicitly enforced in other parallel machines.

[2]We follow the standard definition of *dependence distance vector* as in [Kuhn80].

[3]The details of this transformation are in Section 4.3.

```
For $i = 0$ to $N - 1$
    A: A[i] = f(B[i-1])
    B: B[i] = g(A[i])
    C: C[i] = h(A[i], C[i-1])
Endfor
```



(a) The source code and its dependence graph.

```
For $i = 0$ to $N - 1$
    A: A[i] = f(B[i-1])
    B: B[i] = g(A[i]) C: C[i] = h(A[i],C[i-1])
Endfor
```

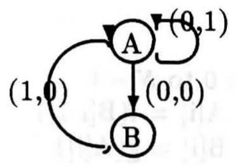
(b) Optimally parallelized form

cycle	schedule
0	A0
1	B0 C0
2	A1
3	B1 C1
4	A2
5	B2 C2
6	. . . .

(c) ASAP schedule

Figure 1: Optimal loop scheduling – one dimensional case.

```
For $i_1 = 0$ to $N_1 - 1$
   For $i_2 = 0$ to $N_2 - 1$
      A:A$[i_1, i_2]$=f(A$[i_1, i_2 - 1]$,B$[i_1 - 1, i_2]$)
      B:B$[i_1, i_2]$=g(A$[i_1, i_2]$)
   endfor
Endfor
```



(a) The source and its dependence graph. In the graph, each edge is associated with a dependence distance vector.

cycle	schedule
0	A00
1	B00 A01
2	B01 A02 A10
3	B02 A03 B10 A11
4	B03 A04 B11 A12 A20
5	B04 A05 B12 A13 B20 A21
6	B05 . B13 A14 B21 A22
7	. . B14 A15 B22 A23
8	. . B15 . B23 .
9	. . . . .

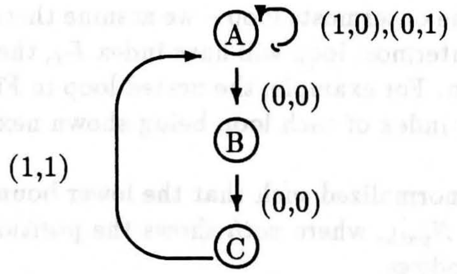
(b) ASAP schedule

```
for $t = 0$ to $(N_1 - 1)2 + (N_2 - 1) + 1$
    forall $i_1 = L_1$ to $U_1$
        forall $i_2 = L_2$ to $U_2$
            case $Max(0, t - 2i_1 - i_2)$ is
                0:$A[i_1, i_2 - 1] = f(A[i_1, i_2 - 1], B[i_1 - 1, i_2])$
                1:$B[i_1, i_2] = g(A[i_1, i_2])$
            endcase
        endfor
    endfor
endfor    ,
```

where $L_1 = Max(0, \lceil (t - N_2)/2 \rceil)$,
$U_1 = Min(N_1 - 1, \lfloor t/2 \rfloor)$,
$L_2 = Max(0, t - 2i_1 - 1)$,
$U_2 = Min(N_2 - 1, t - i_1)$.

(c) The parallel form.

Figure 2: Optimal loop scheduling – 2 dimensional case.



(a) The dependence graph

cycle	schedule											
0	A00											
1	B00	A01			A10							
2	C00	B01	A02		B10				A20			
3		C01	B02	A03	C10	A11			B20			
4			C02	B03		B11	A12		C20	A21		
5				C03		C11	B12	A13		B21		
6							C12	B13		C21	A22	
7								C13			B22	A23
8											C22	B23
9												C23

(b) ASAP schedule

$$d_1(i_1, i_2) = \begin{cases} 2 & \text{if } i_1 < i_2 \\ 1 & \text{if } i_1 \geq i_2 \end{cases}$$

$$d_2(i_1, i_2) = \begin{cases} 2 & \text{if } i_1 > i_2 \\ 1 & \text{if } i_1 \leq i_2 \end{cases}$$

(c) Delay pattern.

Figure 3: A 2-dimensional loop in which the delays are not constant.

The name of each node is represented by $LOOP_{path}$. So, $LOOP_1$ is the root node, $LOOP_{1i}$ is the i_{th} child loop of $LOOP_1$, and so on. We also define the *index* of each node similarly. I_1 is the index of the root node (or the outermosts loop – we assume there is only one outermost loop). The i_{th} child loop of this outermost loop will have index I_{1i}, the j_{th} child loop of the I_i loop will have index I_{1ij}, and so on. For example, the nested loop in Figure 4(a) will have the loop tree in Figure 4(b), with the index of each loop being shown next to the corresponding node.

Also, we assume all loops are normalized such that the lower bounds are always 1's. The upper bounds are represented by N_{path}, where *path* shows the position of the corresponding loop in the loop tree, as in loop indices.

For each node in the loop tree, we define three values: H_{path}, d_{path}, and S_{path}. H_{path} is the number of child loops of $LOOP_{path}$. d_{path} is the amount of delay between the adjacent iterations of $LOOP_{path}$. We will call this value the delay of $LOOP_{path}$. This value exactly corresponds to the *delay* in DOACROSS. Finally, S_{path} is the *size* of loop I_{path}, which is defined as,

$$S_{path} = \begin{cases} (N_{path} - 1)d_{path} + S_{1,p_1,\ldots,p_x,1} + \ldots + S_{1,p_1,\ldots,p_x,H_{path}} & \text{if } LOOP_{path} \text{ is a true loop} \\ 1 & \text{if } LOOP_{path} \text{ is a statement,} \end{cases}$$

when $path = (1, p_1, \ldots, p_x)$.

Note that

$$S_{1,p_1,\ldots,p_x,1} + \ldots + S_{1,p_1,\ldots,p_x,H_{path}}$$

is the sum of the *sizes* of all the child loops of $LOOP_{path}$. We will use a short-hand representation $S_{1,p_1,\ldots,p_x,*}$ for this. For example, the size of $LOOP_{1ij}$ is

$$S_{1ij} = (N_{1ij} - 1)d_{1ij} + S_{1ij*}.$$

These values are needed to transform the loop correctly.

A node in the loop tree is executed a number of times dictated by the upper bounds of its predecessors. For example, $LOOP_{121}$ is repeated by $N_1 \times N_{12}$. The *partial iteration vector* shows which copy is active; that is, it shows the index values of the currently active *surrounding* loops. Therefore, the instance of $LOOP_{121}$ at *partial iteration vector* (iv_1, iv_{12}) is its copy when $I_1 = iv_1$, and $I_{12} = iv_{12}$.

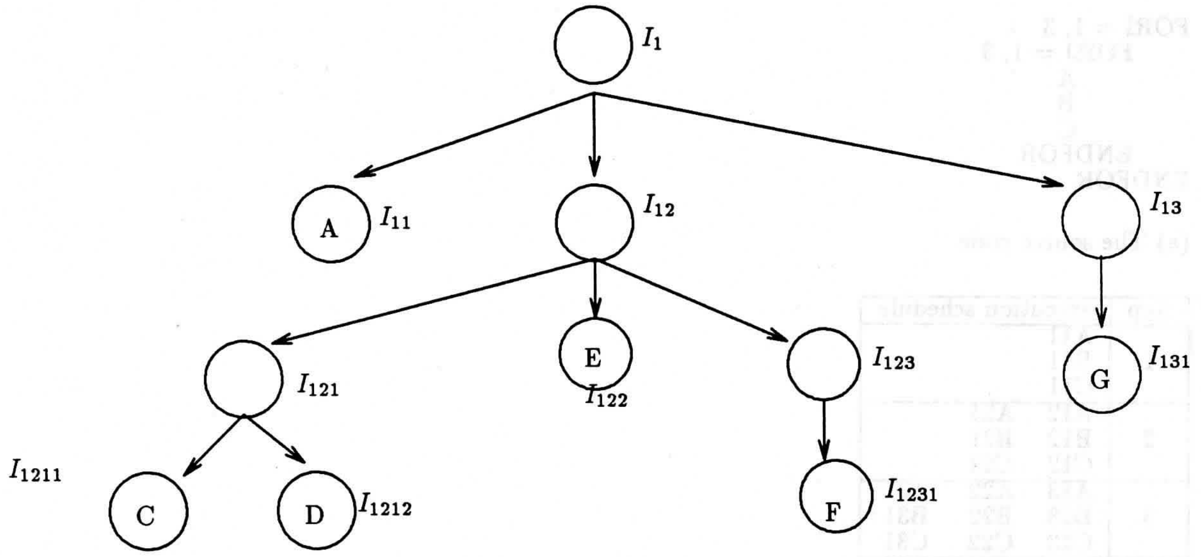
3 Comparison with coarse grain techniques

Our method has some relationships with the Wavefront method suggested in [IrTr88][LaWo90] and DOACROSS [Cytr86]. [LaWo90] present a technique, called Loop Transformation, to transform n nested loops into one sequential outermost loop plus $(n-1)$ parallel inner loops. For example, the loop in Figure 5(a) will be transformed into that in Figure 5(b).

The general technique is to skew the innermost loop (loop K in this example) against all the outer loops, and perform *loop permutation* such that the innermost loop is moved to the outermost position. However, one condition should be met to use this method: the nested loops should be *fully permutable*. The definition of *fully permutable loop nest* and techniques to transform ordinary loops into *fully permutable* ones in part or in whole are in [LaWo90].

```
For $I_1 = 1$ to $N_1$
    A
    For $I_{12} = 1$ to $N_{12}$
        For $I_{121} = 1$ to $N_{121}$
            C
            D
        Endfor
        E
        For $I_{123} = 1$ to $N_{123}$
            F
        Endfor
    endfor
    For $I_{13} = 1$ to $N_{13}$
        G
    Endfor
Endfor
```

(a) An example nested loop



(b) Its loop tree

Figure 4: An example nested loop and its loop tree.

```
FOR$I = 1, N$
    FOR$J = 1, N$
        FOR$K = 1, N$
            loop-body
```

(a) Source

```
FOR$A = L1, U1$
    FORALL $B = L2, U2$
        FORALL $C = L3, U3$
            loop-body
```

(b) Parallel form by Loop Transformation.

Figure 5: Parallel template of Loop Transformation.

```
FORI = 1, 3
    FORJ = 1, 3
        A
        B
        C
    ENDFOR
ENDFOR
```

(a) The source code

step	execution schedule		
1	A11 B11 C11		
2	A12 B12 C12	A21 B21 C21	
3	A13 B13 C13	A22 B22 C22	A31 B31 C31
4		A23 B23 C23	A32 B32 C32
5			A33 B33 C33

(b) The parallel execution schedule of the loops in (a) by Loop Transformation, assuming the loops are already *fully permutable*. The two digits at each statement show the iteration numbers. For example, A23 means statement A for I = 2 and J = 3.

Figure 6: The parallelization by Loop Transformation.

step	execution schedule																	
1	A11	B11																
2	C11		A12	B12			A21	B21										
3			C12		A13	B13	C21		A22	B22			A31	B31				
4					C13				C22		A23	B23	C31		A32	B32		
5											C23				C32		A33	B33
6																	C33	

Figure 7: Another parallel execution schedule where more fine-grain parallelism is exposed.

Our method tries to expose more fine-grain parallelism than the above. Let's see what additional parallelism can be exposed, using an example.

Figure 6(a) is the example loop we will use. Assuming it is already *fully permutable*, Loop Transformation techniques will expose the parallelism as in Figure 6(b). The J loop is skewed by 1 against the I loop and interchanged with the I loop. The parallelism that may exist between the statements, and the parallelism that may exist between different iterations of the J loop is not exposed. Figure 7 shows another schedule where this additional parallelism is exposed, assuming statements A and B can be done at the same time, and A and B at iteration k of J loop can be done at the same time with statement C at iteration $k-1$ of the J loop.

We concentrate on exposing this additional parallelism. In addition, we want our technique of exposing parallelism to be flexible enough to handle non-perfectly-nested loops. Figure 8 shows an example of non-perfectly-nested loop, and the desired parallel schedule we want to obtain.

To do this, we have developed a new method which is different from Loop Transformation in that it does not perform *loop interchange* or *loop permutation.* Instead of performing *loop permutation* to compute the parallel form, it directly computes the parallel form by examining the desired parallel schedule. This feature allows us to expose more fine-grain parallelism even when the loop is non-perfectly-nested.

Another technique that needs to be mentioned is DOACROSS. Looking at Figure 6(b) and Figure 7, we realize similar schedules can be derived from DOACROSS. To get the schedule in Figure 6(b), DOACROSS can be applied to the I loop in the source code (Figure 6(a)), while serializing the J loop. To get the schedule in Figure 7, the loop body would have to be reordered before DOACROSS can be applied. However, DOACROSS does not compute the explicit parallel form to express the parallelism shown in the schedule. Therefore, it does not produce the parallel loop structure as produced by Loop Transformation or our technique (see the parallel form in Figure 13(b) for example).

Furthermore, our technique also differs from DOACROSS in that the parallelism is maximized through *shaping* (explained in Section 4.1). *Shaping* reorders the statements in the loop body while allowing overlapping between them such that the parallelism across *all* dimensions is maximized. Thus there are situations where the parallelism exposed by our technique can not be captured by DOACROSS (even when the loop body is reordered beforehand), and examples do in fact occur in practice, as seen in Section 5.

4 Method

Our method consists of three steps: *shaping, delay computing,* and *transformation.* In the *shaping* process, we reorder the statements to maximize the overall parallelism. In the *delay*

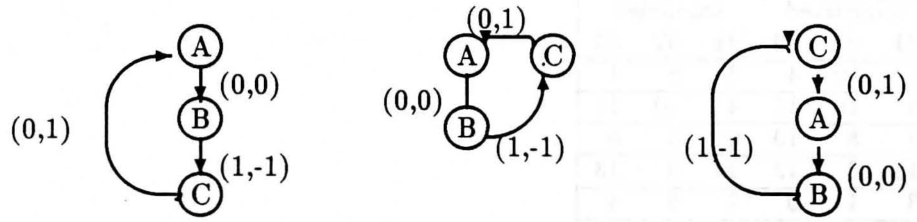
```
FORI = 1, 3
  A
  FORJ = 1,3
    B
    C
  ENDFOR
  FORK = 1,3
    D
    E
  ENDFOR
  F
ENDFOR
```

(a) The source code

step	execution schedule								
1	A1								
2	B11			A2					
3	C11	B12		B21			A3		
4		C12	B13	C21	B22		B31		
5			C13		C22	B23	C31	B32	
6	D11					C23		C32	B33
7	E11	D12		D21					C33
8		E12	D13	E21	D22		D31		
9			E13		E22	D23	E31	D32	
10	F1					E23		E32	D33
11				F2					E33
12							F3		

(b) The desired parallel schedule for the loop in (a).

Figure 8: An example of a non-perfectly-nested loop and its parallel schedule.



$d_1 = 3$ $d_{11} = 3$ (a)Original-order

$d_1 = 3$ $d_{11} = 1$ (b) Compacted-order

$d_1 = 3$ $d_{11} = 0$ (c) Scheduled-order

Figure 9: Three different ordering methods.

computing step, we compute the delays of all participating loops. In the *transformation* step, we actually compute the parallel form.

4.1 Shaping

Shaping is applied to a set of statements within the same loop nest. In perfectly-nested loops, the statements in the innermost loop will become the target. In non-perfectly-nested loops, several set of statements that are surrounded by the same set of loop nests will become the target.

One trivial shaping method is to take the statements in their original order. Let's call this the *original-order method.* A variation of this method is to take the set of statments and schedule it ASAP, ignoring all loop-carried-dependences. This will be referred to as the *compacted-order method.* Neither of them is satisfactory. We take an approach derived from fine-grain scheduling. We unwind the loop a finite number of times for each dimension, schedule all statement instances in it ASAP, and take the order of the statements when a repeating pattern is found. Such a pattern will often occur naturally, and in those cases it in fact expresses the optimal parallelism in the loop; when the pattern does not occur on its own, we simply force it heuristically after a finite number of iterations. [4] This method will be referred to as *scheduled-order method.*

Figure 9(a)-(c) shows a sample loop and the three different orderings. The sample loop is two-dimensional and has three statements (A,B, and C). The dependences between the statments are shown in the figure by the dependence distance vectors. In the figure, we calculated the delay of each loop for each case (see Section 4.2 for the delay computation). d_1 is the delay of the outer loop, and d_{11} is that of the inner loop.

The merit of the *scheduled-order method* is that it rearranges the statements to expose hidden parallelism. For example, in Figure 9(a)-(c), we don't get arbitrarily parallelizable[5]

[4]Note that we perform the unwinding *only* to compute the new ordering; we do not replace the loop body with the unwound loop body.

[5]The parallelism is limited only by the size of the loops.

loops	original			compacted			scheduled		
	d1	d2	d3	d1	d2	d3	d1	d2	d3
1	16	26	33	2	10	4	3	5	4
2	8	22	29	4	10	11	4	10	11
3	38	32	18	8	8	10	0	8	8
4	22	33	16	15	3	19	14	0	13
5	17	9	25	3	1	9	3	0	4
6	16	20	18	1	6	8	4	4	9
7	21	21	12	15	14	14	15	2	10
8	20	9	18	6	3	5	6	2	2
9	19	0	32	7	0	8	3	0	5
10	27	30	9	10	15	9	1	8	6
11	7	25	17	1	7	13	3	6	8
12	29	19	34	15	13	16	0	12	16
13	23	9	17	9	14	12	7	1	12
14	11	13	10	4	9	4	2	9	3
15	20	29	22	4	11	3	3	11	3
16	9	18	9	4	4	5	4	2	5
17	27	25	30	4	10	7	1	10	5
18	15	15	23	1	1	5	0	1	4
19	33	27	34	21	17	23	10	17	22
20	31	20	25	3	7	8	5	1	7

Figure 10: Comparison of the three different methods.

dimensions using the first two methods, while using the third method we can make the first dimension arbitrarily parallelizable (that is $d_{11} = 0$). Since the methods are heuristics, we wanted to test the robustness of their performance under unbiased condition. Thus we have generated 20 random loops, and calculated three different delay sets for each loop. The three delay sets correspond to the above three cases. The loops we have generated contain 50 nodes and 100 edges with nest depth 3. One half of the edges are in-loop dependences, with the remaining dependences being loop carried. The edges are generated randomly. In the case of the third method (*scheduled-order method*), the patterns were found within 10 iterations for all 20 loops. (Fig 10)

The results are presented in Table 10. For loops 3,4,5,12, and 18, the third method detects arbitrary parallelism in some dimensions, while the other two do not. Computing the total sum of delays for each method for each loop, we find the third method produces smaller sums than the other two for all loops except loop 6. For loop 6, the sum of delays obtained by the third method is much smaller than the sum of delays of the first method but slightly larger than that of the second method.

4.2 Delay computing

This step is equivalent (*but not identical*) to the delay computing process in DOACROSS. In general, computing the optimal delay set is an integer programming problem [Lamp74]. The problem can be formulated as follows.

Compute delay set $(d_1, d_2, \ldots, d_y)$ such that it minimizes the total execution time and satisfies the following inequalities.

$$\begin{aligned}
(v_{11}d_1 + v_{12}d_2 + \ldots + v_{1y}d_y) &\geq c_1 \\
(v_{21}d_1 + v_{22}d_2 + \ldots + v_{2y}d_y) &\geq c_2 \\
&\ldots\ldots \\
(v_{x1}d_1 + v_{x2}d_2 + \ldots + v_{xy}d_y) &\geq c_x, \qquad (1)
\end{aligned}$$

where y is the cardinality of the delay set, and x is the number of dependence distance vectors.

In the above, $v_{i1}, v_{i2}, \ldots, v_{iy}$ is the i_{th} dependence distance vector, and c_i is some constant. Note that to simplify the notations we gave each delay a serial number; so, the subscripts of d here do not represent the *path* as we did in Section 2.

One interesting observation is that if one of the v column contains only positive elements (greater than 0), say in the j_{th} column, then we can solve the system simply by putting $d_k = 0$ for all $k \neq j$, and putting d_j = some integer that satisfies the remaining system. Based on this observation, plus the fact that the first non-zero element of a legal dependence distance vector should be positive, we can suggest a delay computing algorithm which is much simpler but still more effective compared to previous approaches such as [Cytr86].

Basically, the algorithm works in divide-and-conquer manner. The inequalities are divided into two groups: those whose leading coefficients are positive – call this the first group; and those whose leading coefficients are zero's – call this the second group. The second group is solved first. Note that this problem has one less variables than the original one. The delay corresponding to the leading coefficient is missing here. Assume the second group is solved, which means we know all the values of the delays except the one corresponding to the leading coefficient. We substitute these delay values to the first group of inequalities (whose leading coefficients are positive), and compute the missing delay. The solving process of the second group is a recursive application of the same divide-and-conquer strategy. Please refer to [KiNi91] for the details of this algorithm.

4.3 Transformation

The third step transforms the *loop tree* into a *parallel loop tree* according to the delays computed in the previous section. All nodes (except the root and leaf nodes) in the loop tree are transformed into parallel form as follows. Suppose we want to transform $LOOP_{1ij}$ into parallel form. Assume it has three child loops. Then, $LOOP_{1ij}$ below,

```
.....
.....
 For $I_{1ij}$ = 1 to $N_{1ij} - 1$
     For $I_{1ij1}$ = 1 to $N_{1ij1} - 1$
     ....
     Endfor
     For $I_{1ij2}$ = 1 to $N_{1ij2} - 1$
     ....
     Endfor
```

```
      For $I_{1ij3} = 1$ to $N_{1ij3} - 1$
          ....
      Endfor
  Endfor
...... ......
```

will be transformed into

```
.....
.....
Forall $I_{1ij} = L_{1ij}$ to $U_{1ij}$
      Case $C_{1ij}$ is
      1 to $S_{1ij1}$ : For $I_{1ij1} = 1$ to $N_{1ij1}$
                            ....
                        Endfor
      $S_{1ij1} + 1$ to $S_{1ij1} + S_{1ij2}$ : For $I_{1ij2} = 1$ to $N_{1ij2}$
                                                 ....
                                             Endfor
      $S_{1ij1} + S_{1ij2} + 1$ to $S_{1ij1} + S_{1ij2} + S_{1ij3}$ : For $I_{1ij3} = 1$ to $N_{1ij3}$
                                                                 ....
                                                             Endfor
      Endcase
Endforall
......
......
```

Note that the inner loops of $LOOP_{1ij}$ are not parallelized yet. They can be parallelized by applying the same process recursively. Several new notations are used. L_{1ij} and U_{1ij} are the new loop bounds; their computation will be explained later. S_{path} is the size of $LOOP_{path}$ as explained in Section 2. C_{1ij} is the range variable for the case statement, which is defined as

$$C_{1ij} = t_{1ij} - (I_{1ij} - 1)d_{1ij}.$$

Here t_{1ij} is the *local time step* of $LOOP_{1ij}$, whose value is computed by

$$t_{1ij}(I_1 = a, I_{1i} = b) = t - SC_{1ij}(I_1 = a, I_{1i} = b) + 1,$$

where t is the *global time step*, and $SC_{1ij}(I_1 = a, I_{1i} = b)$ is the starting time step (or starting cycle) of the instance of $LOOP_{1ij}$ at partial iteration vector $(I_1 = a, I_{1i} = b)$ (see Section 2 for the definition of partial iteration vector). So, $t_{1ij}(I_1 = a, I_{1i} = b)$ represents the time elapsed since the instance of $LOOP_{1ij}$ at partial iteration vector $(I_1 = a, I_{1i} = b)$ began – the *local time step.* On the other hand t represents the time elapsed since the entire loop began execution; so it is called *global time step.*

The computation of SC_{1ij} proceeds as follows. Assume we want to compute SC_{1ij} at partial iteration vector $(I_1 = iv_1, I_{1i} = iv_{1i})$. If we draw the surrounding loops of $LOOP_{1ij}$, we get

```
For $I_1 = 1$ to $N_1$
    For $I_{11} = 1$ to $N_{11}$
        ...
    Endfor
    For $I_{12} = 1$ to $N_{12}$
        ...
    Endfor
    ...
```

```
...
For $I_{1i} = 1$ to $N_{1i}$
    For $I_{1i1} = 1$ to $N_{1i1}$
    ...
    Endfor
    For $I_{1i2} = 1$ to $N_{1i2}$
    ...
    Endfor
    ...
    ...
    For $I_{1ij} = 1$ to $N_{1ij}$
    ...
    Endfor
    ...
    ...
Endfor
...
...
Endfor.
```

We want to calculate the starting time step of the instance of $LOOP_{1ij}$ when $I_1 = iv_1$, and $I_{1i} = iv_{1i}$. Since each iteration of $LOOP_1$ is delayed by d_1, the iv_1-th iteration of $LOOP_1$ will start at $(iv_1 - 1)d_1$ time step. At iv_1-th iteration of $LOOP_1$, we have to wait until all sub-loops preceding $LOOP_{1i}$ are executed. Therefore, $S_{11} + \ldots + S_{1,i-1}$ time steps should be passed. At this point, we again have to wait for the iv_{1i}-th iteration of $LOOP_{1i}$. This adds $(iv_{1i} - 1)d_{1i}$ times steps to the delay time accumulated so far. Finally, we have to wait until all the sub-loops before $LOOP_{1ij}$ at the iv_{1i}-th iteration are executed. So, the starting time step of the desired instance of $LOOP_{1ij}$ is

$$SC_{1ij}(I_1 = iv_1, I_{1i} = iv_{1i}) = (iv_1 - 1)d_1 + S_{11} + \ldots + S_{1,i-1} + (iv_{1i} - 1)d_{1i} + S_{1i1} + \ldots + S_{1,i,j-1} + 1.$$

Now, we will explain how to compute L_{1ij} and U_{1ij}, the new loop bounds. L_{1ij} is the iteration that spans t_{1ij}, the local time step of the current instance of $LOOP_{1ij}$, for the first time. U_{1ij} is the last iteration that spans t_{1ij}. Therefore, if $L_{1ij} > 1$, the ending time step[6] of the iteration $L_{1ij} - 1$ should be strictly less than t_{1ij}, and the ending time step of the iteration L_{1ij} should be greater than or equal to t_{1ij}. Also if $U_{1ij} < N_{1ij}$, the starting time step of U_{1ij} should be greater than or equal to t_{1ij}, while that of $U_{1ij} + 1$ should be strictly greater than t_{1ij}. Therefore, when $L_{1ij} > 1$ and $U_{1ij} < N_{1ij}$, we get the following inequalities to be satisfied.

$$(L_{1ij} - 2)d_{1ij} + S_{1ij*} < t_{1ij} \leq (L_{1ij} - 1)d_{1ij} + S_{1ij*}$$

$$(U_{1ij} - 1)d_{1ij} + 1 \leq t_{1ij} < U_{1ij}d_{1ij} + 1$$

Solving these with the constraints that L_{1ij} is an integer greater than or equal to 1, and U_{1ij} is an integer less than or equal to N_{1ij}, we get

$$L_{1ij} = MAX(1, \lceil \frac{t_{1ij} - S_{1ij*}}{d_{1ij}} + 1 \rceil),$$

[6]Actually local ending time step. We are looking at only the current instance of $LOOP_{1ij}$. Every time step here, while we are explaining the computation of L_{1ij} and U_{1ij}, refers to the local time step of the current instance of $LOOP_{1ij}$.

and

$$U_{1ij} = MIN(N_{1ij}, \lfloor \frac{t_{1ij} - 1}{d_{1ij}} + 1 \rfloor).$$

Now, the same parallelization process can be repeated for all the intermediate nodes. The parallelization of the leaf nodes is simple: just leave them untouched. For the root node (the outermost loop), we parallelize it following the above process, but this time add another loop on top of it. The new outermost loop is a sequential loop, and its index is the global time step (t). At each global time step, the sequential outermost loop specifies which statements of which loops can be executed in parallel.

Using these notations, the parallel template of the loop in Figure 11(a), for example, will become that in Figure 11(b).

The general formulas for transformation are summarized in Figure 12. We assumed $path = 1ijk$ to simplify the notations. It should be easily extendible to other paths. Note that $SC_{1ijk} = 1$ when ijk is nil, because $SC_1 = 1$ by definition (it is the starting time step of the outermost loop). For completeness, we have included the formula for MAX-GLOBAL-TIME-STEP in the figure. MAX-GLOBAL-TIME-STEP is the number of total time steps needed to execute the given loop.

4.4 An example

An example loop is given in Figure 13. In the figure, we show the source code, the parallel template, and the loop tree.

Let's assume that $d_1 = d_{11} = d_{111} = 1$, and that after ***shaping*** we got the statement arrangement as seen in Figure 13(b) (note that statement C is moved up next to A).

Let's first compute the sizes of all loops. From the formulas in Figure 12,

$$S_{111} = (3-1)d_{111} + S_{1111} + S_{1112} = 4.$$

$$S_{11} = (3-1)d_{11} + S_{111} = 6,$$

and

$$S_1 = (3-1)d_1 + S_{11} = 8.$$

Now,

$$SC_1 = 1,$$

$$SC_{11} = (I_1 - 1)d_1 + 1 = I_1,$$

and

$$SC_{111} = (I_1 - 1)d_1 + (I_{11} - 1)d_{11} + 1 = I_1 + I_{11} - 1.$$

Then,

$$t_1 = t - SC_1 + 1 = t,$$

$$t_{11} = t - SC_{11} + 1 = t - I_1 + 1,$$

and

$$t_{111} = t - SC_{111} + 1 = t - I_1 - I_{11} + 2.$$

Therefore,

$$C_{111} = t_{111} - (I_{111} - 1)d_{111} = t - I_1 - I_{11} - I_{111} + 3.$$

```
For $I_1 = 1$ to $N_1$
    A
    For $I_{12} = 1$ to $N_{12}$
        For $I_{121} = 1$ to $N_{121}$
            C
            D
        Endfor
        E
        For $I_{123} = 1$ to $N_{123}$
            F
        Endfor
    endfor
    For $I_{13} = 1$ to $N_{13}$
        G
    Endfor
Endfor
```

(a) The source code.

```
For $t = 1$ to MAX-GLOBAL-TIME-STEP
    Forall $I_1 = L_1$ to $U_1$
        Case $C_1$ is
        $1$ : A
        $2$ to $1 + S_{12}$ :
            Forall $I_{12} = L_{12}$ to $U_{12}$
                Case $C_{12}$ is
                $1$ to $S_{121}$:
                    Forall $I_{121} = L_{121}$ to $U_{121}$
                        Case $C_{121}$ is
                        1: C
                        2: D
                        Endcase
                    Endforall
                $S_{121} + 1$: E
                $S_{121} + 2$ to $S_{121} + 1 + S_{123}$:
                    Forall $I_{123} = L_{123}$ to $U_{123}$
                        F
                    Endforall
                Endcase
            Endforall
        $S_{12} + 2$ to $S_{12} + 1 + S_{13}$:
            Forall $I_{13} = L_{13}$ to $U_{13}$
                G
            Endforall
        Endcase
    Endforall
Endfor
```

(b) Its parallel template.

Figure 11: An example of parallel template.

$$SC_{1ijk}(I_1 = a, I_{1i} = b, I_{1ij} = c) =$$

$$\begin{cases} 1 & \text{if the path } ijk \text{ is nil} \\ (a-1)d_1 + S_{11} + S_{12} + \ldots + S_{1,i-1} & \\ +(b-1)d_{1i} + S_{1i1} + S_{1i2} + \ldots + S_{1,i,j-1} & \\ +(c-1)d_{1ij} + S_{1ij1} + S_{1ij2} + \ldots + S_{1,i,j,k-1} + 1 & \text{if the path } ijk \text{ is not nil} \end{cases}$$

$$L_{1ijk} =$$

$$\begin{cases} MAX(1, \lceil \frac{t_{1ijk} - S_{1ijk*}}{d_{1ijk}} + 1 \rceil) & \text{if } d_{1ijk} > 0 \\ 1 & \text{if } d_{1ijk} = 0 \end{cases}$$

$$U_{1ijk} =$$

$$\begin{cases} MIN(N_{1ijk}, \lfloor \frac{t_{1ijk} - 1}{d_{1ijk}} + 1 \rfloor) & \text{if } d_{1ijk} > 0 \\ N_{1ijk} & \text{if } d_{1ijk} = 0 \end{cases}$$

$$t_{1ijk} = t - SC_{1ijk} + 1$$

$$C_{1ijk} = t_{1ijk} - (I_{1ijk} - 1)d_{1ijk}$$

$$S_{1ijk} = \begin{cases} (N_{1ijk} - 1)d_{1ijk} + S_{1ijk*} & \text{if } LOOP_{1ijk} \text{ is a loop} \\ 1 & \text{if } LOOP_{1ijk} \text{ is a statement.} \end{cases}$$

$$\text{MAX-GLOBAL-TIME-STEP} = S_1$$

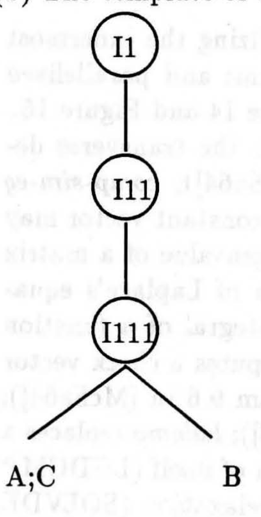
Figure 12: Transformation formula.

```
FOR $I_1 = 1,3$
    FOR $I_{11} = 1,3$
        FOR $I_{111} = 1,3$
            A
            B
            C
```

(a) The source code

```
FOR $t = 1, MAX\_GLOBAL\_TIME\_STEP$
    FORALL $I_1 = L_1, U_1$
          FORALL $I_{11} = L_{11}, U_{11}$
                FORALL $I_{111} = L_{111}, U_{111}$
                CASE $C_{111}$ is
                      1: A; C
                      2: B
                ENDCASE
```

(b) The template of the parallel loop



(c) The loop tree

Figure 13: An example loop.

Finally,

$$L_1 = MAX(1, t_1 - S_{11} + 1) = MAX(1, t - 5),$$
$$L_{11} = MAX(1, t_{11} - S_{111} + 1) = MAX(1, t - I1 - 2,$$
$$L_{111} = MAX(1, t_{111} - 2 + 1) = MAX(1, t - I_1 - I_{11} + 1),$$
$$U_1 = MIN(3, t),$$
$$U_{11} = MIN(3, t - I_1 + 1),$$

and

$$U_{111} = MIN(3, t - I_1 - I_{11} + 2.$$

Therefore, the fully instantiated parallel form is

```
FOR $t = 1,8$
  FORALL $I_1 = MAX(1,t-5), MIN(3,t)$
      FORALL $I_{11} = MAX(1,t-I_1-2), MIN(3,t-I_1+1)$
          FORALL $I_{111} = MAX(1,t-I_1-I_{11}+1), MIN(3,t-I_1-I_{11}+2)$
          CASE $t-I_1-I_{11}-I_{111}+3$ is
              1: A; C
              2: B
          ENDCASE
```

5 Experiments

To see the benefits of parallelizing nested loops as opposed to parallelizing the innermost loop only, we have collected 10 loops from various numerical algorithms and parallelized them using our method and Perfect Pipelining. The results are in Figure 14 and Figure 15.

A brief explanation of each algorithm follows. *2d-bnd-val* computes the transverse deflections of a simply-supported rectangular plate (program 8.4 in [McSa64]); *comp-sim-eq* solves a system of linear simultaneous equations whose coefficients and constant vector may be complex (program 9.5 in [McSa64]); *eig-val* computes the largest eigenvalue of a matrix by iteration (program 5.5 in [McSa64]); *laplace* computes the solution of Laplace's equation by iteration (program 8.1 in [McSa64]); *simpson* evaluates the integral of a function using Simpson's 1/3 rule (program 9.9-A in [McSa64]); *sub-comeqs* computes a check vector by substituting the solution vector into the original equations (program 9.6 in [McSa64]); *vpenta* inverts three pentadiagonals simultaneously (VPENTA in [Bail88]); *ludcmp* replaces a given $N \times N$ matrix by the LU decomposition of a rowwise permutation of itself (LUDCMP in [PFTV86]); *solvde* solves two point boundary value problems by relaxation (SOLVDE in [PFTV86]); and *pinvs* diagonalizes the square subsection of a given matrix (PINVS in [PFTV86]).

Figure 14 compares 4 values: the execution time of sequential, Perfect Pipelining, DOACROSS, and N-Dimensional Perfect Pipelining. The execution time is measured assuming that each statement takes 1 unit time and that sufficient resources are available to the advantage of the exposed parallelism. The problem size of each case is also shown in the table. l represents

loop	seqential	PP	DOACROSS	NDPP	problem size
2d-bnd-val	149020	37020	2622	114	$l = 10, n = 100, m = 1456$
comp-sim-eq	2430000	81000	2700	2700	$n = 30$
eig-val	2001000	3000	1002	1002	$n = 1000$
laplace	4000000	3000000	1489	1191	$l = n = 100$
simpson	2252400	183900	5416	3621	$n = 300$
sub-comeqs	8002000	1004000	1009	1009	$n = 1000$
vpenta	27000000	8000	23004	6002	$n = 1000$
ludcmp	2006000	1004000	4004	3002	$l = n = 1000$
solvde	5007000	2001000	3009	3009	$l = n = 1000$
pinvs	2004000	2004000	3003	3003	$l = n = 1000$

Figure 14: Comparison of execution time between sequential, Perfect Pipelining (PP), DOACROSS, and N-Dimensional Perfect Pipelining (NDPP).

the size of the outmost loop, n that of the next level loop, and m that of the innermost loop. When the size of all loops are the same, we use n to represent one of them. We picked the problem sizes such that the sequential execution times are order of 10^6.

The speed-ups (over sequential) obtained by PP, DOACROSS, NDPP are summarized in Figure 15. Since PP works only on the innermost loop, it shows much smaller speed-ups than the other two technique in general. We observe NDPP shows better speed-ups than DOACROSS half the loops, and equal speed-ups for the rest of them. However, this tells only part of the story. By untilizing further fine grain parallelism, all the results for NDPP could be improved more. Of practical interest is *2d-bnd-val* where clearly NDPP is finding surprisingly more parallelism than could be obtained by combining DOACROSS with Perfect Pipelining. Another difference is that the way of exposing parallelism in NDPP and DOACROSS are different: NDPP is explicit, DOACROSS implicit. This difference leads to two different architectures that each technique is better suited; NDPP prefers synchronous multiprocessors (equipped with global barriers) with each processing element being a VLIW, while DOACROSS is more natural for asynchronous multiprocessors. This point is the topic of the next section.

6 Mapping

The ideal architecture to exploit the parallelism exposed by our method would be a multiprocessor in which each processing element is a VLIW machine. It is desirable that the multiprocessor has a global barrier synchronization mechanism.

Let's take the parallel form in Figure 13(b) to see how it can be mapped to such an architecture. The parallel form has one sequential outermost loop and three inner parallel loops. For each t (the index of the outermost loop), the three inner loops define a different iteration space, and all points in this iteration space can be done in parallel if resources permitting. A point in this space corresponds to an instance of the case statement shown in the loop body of Figure 13(b).

Then, the iteration space defined by the three inner loops will be mapped to the set

loop	PP	DOACROSS	NDPP
2d-bnd-val	4.0	57	1307
comp-sim-eq	30	900	900
eig-val	667	1997	1997
laplace	1.3	2686	3358
simpson	12	415	622
sub-comeqs	8	7930	7930
vpenta	3375	1173	4498
ludcmp	2	501	668
solvde	2.5	1664	1664
pinvs	1	667	667

Figure 15: The speed-up table of PP, DOACROSS, and NDPP.

of processing elements, and each processing element will execute the corresponding case statment. The set of processing elements will execute the iteration space defined at each t. The execution of the current iteration space is separated from the next iteration space by the global barrier.

The case statement consists of a test and a set of statements. Depending on the result of the test, a subset of statements are selected to be executed. The statements inside this subset can also be done in parallel; this parallelism is exploited by the multiple functional units each VLIW element has.

The parallelism that each case statement contains is due to the *shaping* process (see Section 4.1). The *shaping* process overlaps statements in order to maximize the parallelism across all loops; the parallelism across iterations is exploited by the multiple processing elements, and the parallelism between statements is exploited by the multiple functional units inside each processing element. The parallelism across iterations exposed by our technique is often much greater than that by DOACROSS because we *shape* the statements inside the loop body in order to minimize the delay in each dimension (and maximize the parallelism across dimensions).

The parallelism in the case statement can be increased if enough parallelism exists in the loops. It will be useful when there is not enough parallelism for the single processing element, as in this particular example. This is achieved by combining Perfect Pipelining and loop unwinding in our transformation. In fact, an advantage of our technique is that it allows *trade-offs* between the coarse and fine grain parallelism "levels" to be easily made. We first apply Perfect Pipelining to the innermost loop, then apply N-Dimensional Perfect Pipelining to the modified loop, and then unwind the innermost loop of the parallel loop k times. By increasing the value of k, we can increase the amount of parallelism under the case statements.

An example is given in Figure 16. In the figure, we assume that $d_1 = d_2 = d_3 = 1$, and that there are strict dependences between A and B, between B and C, and between C and D; this will prevent us from getting any parallelism through *shaping*. Since *shaping* does not produce any parallelism, we have a degenerated case. We will show how we can increase the parallelism under the case statement in this case.

The first two loops in Figure 16 show the original sequential code and its perfect-pipelined version. The third loop shows the parallel form when N-Dimensional Perfect Pipelining is applied to the second loop. We show only the parallel template to explain the idea more clearly.

As we can see in this parallel form, the number of statements that can be done in parallel at each case statement is 1 to 4. However, in most cases we are executing 4 statements in parallel, because $C2$ will be in the range of $(4, N3)$ most of time. This is already an improvement over the parallel form that may be produced by the regular N-Dimensional Perfect Pipelining. In the regular parallel form that N-Dimensional Perfect Pipelining may produce for this example, only one statement would be executed under each case statement because of the strict dependences between statements.

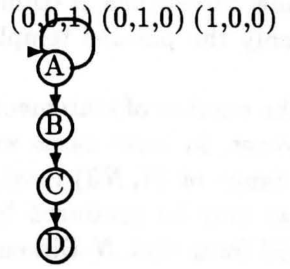
Further improvement of the parallelism under the case statements is possible through loop unwinding. The fourth loop in Figure 16 shows the parallel form after unwinding the parallel $I2$ loop in Figure 16(c) twice (we omitted the parenthesis showing the I3 value for each statement). The case statement here (in Figure 16(d)) examines the value of $C2(I2)$ and $C2(I2+1)$ at the same time, as $I2$ iterates from $L2$ to $U2$ with stride 2. We know from Figure 16(c) that $C2(I2)$ can have 7 different values when $I2$ iterates from $L2$ to $U2$. So, the combination of $C2(I2)$ and $C2(I2+1)$ could lead to $7 \times 7 = 49$ different cases. However, because the values of $C2(I2)$ and $C2(I2+1)$ are not independent of each other, we see only 9 different cases as seen in Figure 16(d). In this improved parallel form, we see 1 to 8 parallel statements under each case statement with 8 statements being the most frequent one.

7 Conclusion

In this paper, we have introduced a new technique to parallelize nested loops at the fine grain level. Previously, parallelizing loops at the fine grain level was largely limited to single-nested loops.

The technique consists of three parts: *shaping, delay computing,* and *transformation.* In *shaping*, we reorder the statements to maximize the parallelism. A heuristic to find a good reordering is suggested. In *delay computing*, we determine the delay of each loop nest based on the above reordering. The technique of DOACROSS might be used in this phase to determine the delays. However, we suggest a simpler and more efficient method to compute the delays. In *transformation*, we compute the parallel form based on the delay of each loop nest. Our transformation technique is powerful and flexible; it exposes more parallelism than previous techniques and applies to general nested loops.

```
FOR $I1 = 1, N1$
  FOR $I2 = 1, N2$
    FOR $I3 = 1, N3$
      A
      B
      C
      D
```



(a) Source code and its dependence graph.

```
FOR $I1 = 1, N1$
  FOR $I2 = 1, N2$
    $A(I3 = 1)$
    $B(I3 = 1)\ A(I3 = 2)$
    $C(I3 = 1)\ B(I3 = 2)\ A(I3 = 3)$
    FOR $i3 = 1, N3 - 3$
      $D(I3 = i3)\ C(I3 = i3 + 1)\ B(I3 = i3 + 2)\ A(I3 = i3 + 3)$
    $D(I3 = N3 - 2)\ C(I3 = N3 - 1)\ B(I3 = N3)$
    $D(I3 = N3 - 1)\ C(I3 = N3)$
    $D(I3 = N3)$
```

(b) Perfect Pipelining is applied to the innermost loop. The expression inside the parenthesis shows the copy of the statment at some value of $I3$. For example, $A(I3 = 3)$ implies the copy of A when $I3 = 3$.

```
FOR $t = 1, MAX\_GLOBAL\_TIME\_STEP$
  FORALL $I1 = L1, U1$
    FORALL $I2 = L2, U2$
    CASE $C2(I2)$ is
       1: $A(I3 = 1)$
       2: $B(I3 = 1)\ A(I3 = 2)$
       3: $C(I3 = 1)\ B(I3 = 2)\ A(I3 = 3)$
       4 to N3: $D(I3 = i3)\ C(I3 = i3 + 1)\ B(I3 = i3 + 2)\ A(I3 = i3 + 3)$
       $N3 + 1$: $D(I3 = N3 - 2)\ C(I3 = N3 - 1)\ B(I3 = N3)$
       $N3 + 2$: $D(I3 = N3 - 1)\ C(I3 = N3)$
       $N3 + 3$: $D(I3 = N3)$
    ENDCASE
```

(c) N-Dimensional Perfect Pipelining is applied to the loop in (b).

```
FOR t = 1, MAX_GLOBAL_TIME_STEP
   FORALL I1 = L1, U1
      FORALL I2 = L2, U2 by 2
      CASE C2(I2) and C2(I2+1) is
      1,*                 : A
      2,1                 : B A     A
      3,2                 : C B A   B A
      4,3                 : D C B A C B A
      5 o N3, 4 to N3-1   : D C B A D C B A
      N3+1,N3             : D C B   D C B A
      N3+2,N3+1           : D C     D C B
      N3+3,N3+2           : D       D C
       *,N3+3             :         D
```

(d) Unwinding the loop in (c) by 2. * means *don't care*.

Figure 16: Increasing the parallelism under the case statement.

References

[AiNi88] Aiken,A. and Nicolau,A., *Perfect Pipelining: a new loop parallelization technique*, Proc. of the 1988 European Symposium on Programming, pp 221-235, Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.

[AlKe87] Allen,J.R. and K.Kennedy, *Automatic Translation of Fortran Programs to Vector Form*, ACM Transctions on Programming Languages and Systems, Vol. 9, No. 4, pp.491-542, Oct. 1987.

[Bail88] Bailey, D.H., NAS kernel benchmark program, NASA Ames Research Center, June 24, 1988.

[Bane90] Banerjee,U., *Unimodular Transformations of Double Loops*, 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August, 1990.

[CCK87] Callahan, D., Cocke, J., and Kennedy, K., *Estimating Interlock and Improving Balance.*, Proc. of the 1987 International Conf. on Parallel Processing, pp 295-304, August, 1987.

[Cytr86] Cytron,R.G., *Doacross: Beyond Vectorization for Multiprocessors.* Proc. of the 1986 International Conf. on Parallel Processing, St. Charles, Ill, pp 836-844, August, 1986.

[Fish79] Fisher, J.A. *The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources.* Ph.D. thesis, New York Univ., 1979.

[GrLa86] Gross, T. and Lam, M., *Compilation for a high-performance systolic array.*, Proc. for the SIGPLAN 1986 Symposium on Compiler Construction, July 1986.

[IrTr88] Irigoin, F. and Triolet, R., *Supernode partitioning,* Proc. of 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, January 1988.

[Kenn80] Kennedy, K., *Automatic Translation of Fortran Programs to Vector Form*, Rice Technical Report 476-029-4, Rice University, Houston, Oct. 1980.

[KiNi91] Kim,K.C. and Nicolau,A., *Fine Grain Software Pipelining of Non-Vectorizable Nested Loops,* Proc. of the International Symposium on Shared Memory Multiprocessing, 1991.

[KKPL81] Kuck,D.J., Kuhn, R.H., Padua, D.A., Leasure, B., and Wolfe, M., *Dependence Graphs and Compiler Optimization*, Proc of the 8th ACM Symp on Programming Languages, Williamsburg, VA, pp. 207-218, Jan. 1981.

[Kuhn80] Kuhn, R.H., *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage networks, and Decision Trees,* Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 80-1009, (UMI 80-26541), Feb. 1980.

[Lam87] Lam, M., *A Systolic Array Optimizing Compiler.* Ph.D. thesis, Carnegie Mellon Univ., 1987.

[Lamp74] Lamport, L., *The Parallel Execution of DO Loops*, Comm. of the ACM, pp.83-93, Feb. 1974.

[LaWo90] Lam, M. and Wolf, M., *Maximizing Parallelism Via Linear Loop Transformations*, 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August, 1990.

[McSa64] McCormick, J.M. and Salvadori M.G., *Numerical Methods in Fortran*, Prentice Hall Inc., 1964.

[Mura71] Muraoka, Y., *Parallelism Exposure and Exploitation in Programs*, Ph.D. These, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 71-424, Feb. 1971.

[Nico85] Nicolau, A., *Uniform Parallelism Exploitation in Ordinary Programs.* Proc. International Conf. on Parallel Processing, August 1985.

[Nico87] Nicolu, A., *Loop Quantization or Unwinding Done Right.* Proc. Supercomputing 1st International Conference, June 1987.

[PFTV86] Press,W.H.,Flannery,B.P.,Teukolsky,S.A., and Vetterling,W.T., *Numerical Recipes*, pp.647-659, Cambridge university press, 1986.

[Wolf82] Wolfe, M. J., *Optimizing Supercompilers for Supercomputers*, Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 82-1105, Oct. 1982.