A Systematical Partitioning Mechanism for Nested Loops with Non-uniform Dependences

Der-Lin Pean, Guan-Joe Lai[*] and Cheng Chen

Department of Computer Science and Information Engineering,National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

[*] Department of Elementary Education, National Taichung Teachers College, Taichung, Taiwan, R.O.C.

E-mail: cchen@csie.nctu.edu.tw

ABSTRACT

This paper proposes a generalized and systematical mechanism to exploit parallelism from nested loops with non-uniform dependences. This mechanism could partition all kinds of non-uniform dependence loops effectively. Our approach, based on the dependence convex theory and the optimized integer programming technique, divides loops into parallel partitions with variable size. All non-uniform dependence loops could be classified into four types. Adaptive mechanisms are proposed for each type to minimize the number of parallel regions in the iteration space. Consequently, our mechanism outperforms previous ones not only in the number of parallel partitions but also in the performance in the real machine.

Keywords: *non-uniform dependence, loop parallelization, parallel compiler and dependence convex hull.*

1. INTRODUCTION

Loop dependences could be classified into two categories: uniform and non-uniform dependences [1]. A pattern of dependence vectors, which are expressed by constants, will be known as uniform dependence vectors. Other dependence vector patterns, which cannot be expressed by constants, belong to non-uniform dependences. Fig. 1 explicates non-uniform and uniform dependence loops.

```
(a)
for I=1, 10
  for J=1, 10
     A (2*I+3, J+1) =...
     ...= A (2*J+I+1, I+J+3)
  endfor
endfor

(b)
for I=1, 10
  for J=1, 10
     A (I,J) = .....
     ...= A(I+1,J)+A(I-1,J)
          +A(I,J+1)+A(I,J-1)
  endfor
endfor
```

Fig. 1 (a) A non-uniform dependence loop, (b) a uniform dependence loop.

Because there is a rich parallelism of loops in scientific programs, parallel compilers have been written to exploit the parallelism [1]. Several techniques based on the convex hull theory have been proposed, such as dependence uniformization [18], minimum dependence distance tiling [2, 11-12, 16-17], three-region partition [21], unique set oriented partitioning [5] and improved three-region partitioning (ITRP) [3]; and most of them fail in parallelizing nested loops with non-uniform dependences. However nearly 45 % of two-dimensional array references are coupled [13], and most of them generate non-uniform dependences. Therefore, a systematical partitioning scheme for parallelizing loops with non-uniform dependences is proposed here. Our method could extract parallelism from nested loops with all kinds of dependence patterns efficiently. Non-uniform dependence loops could be classified into four categories according to dependence vector patterns, the size of dependence convex hull and dependence vector lines. Then adaptable mechanisms could be applied to specific types of non-uniform dependence loops.

Our mechanism is briefly described as follows. First, if special dependence vector patterns with higher parallelism degree could be found in non-uniform dependence loops, we will detect and partition the iteration space according to the special patterns. Second, we classify loops according to the size of the area that could not be parallelized by using the information of dependence convex hulls and dependence vector lines of the iteration space. We call this area the inherent serial area (ISA). If the ISA does not exist in the iteration space, by using our optimized three-region partitioning (OTRP) technique [6, 10], the iteration space could be partitioned into two partitions in which iterations could be executed in parallel. Otherwise, if the ISA exists and its size does not cover all of the iteration space, we could apply our two stage partitioning (TSP) mechanism [9]. TSP mechanism partitions the iteration space into two parallel areas and one serial area at stage one; and at stage two, it partitions the serial region based on integer programming, dependence convex hulls and dependence vector lines. Finally, we propose two mechanisms, partial parallelization decomposition (PPD) [7] and optimized dependence convex hull partitioning (ODCHP) [8-9], to partition the iteration space in which the ISA covers all of the loop iteration space. PPD decomposes the iteration space by their partial minimum dependence distances; and ODCHP partitions the iteration space according to dependence vector lines, dependence convex hulls and related techniques. They also can be combined with our loop restructuring mechanisms such as parallelization part splitting (PPS) [7] and irregular loop interchange (ILI) [7] to exploit parallelism.

Four popular program models and two kernel code segments in real programs are evaluated by our scheme in a CONVEX SPP-1000 with 8 processors, and a multiprocessor system environment called SEESMA [14]. Experimental results show that our scheme performs better than existing ones such as uniformization, minimum dependence distance tiling and ITRP.

The rest of this paper is organized as follows. Section 2 describes several related work. Section 3 presents the categorization of non-uniform dependence loops and partition mechanisms. The preliminary performance results are illustrated in Section 4. Finally, we give some conclusions with suggestions for future work.

2. PRELIMINARIES AND RELATED WORK

Most loops with complex array subscripts are two-dimensional loops [13]. For a simplification of the explanation, the considered program models in this paper are doubly nested loops with coupled subscripts. The solution to multilevel nested loops could be obtained by enhancing our mechanisms. A doubly nested loop model is depicted in Fig. 2, where $f_1(I,J)$, $f_2(I,J)$, $f_3(I,J)$, and $f_4(I,J)$ are linear functions of loop variables. The dimension of the nested loop is equal to the number of nested loops. In loop $I(J)$, $L_I(L_J)$ and $U_I(U_J)$ indicate the lower and upper bounds, respectively. Both the lower and upper bounds of indices should be known at compile time.

```
for I=L_I,U_I
    for J=L_J,U_J
        …
        S_d:  A(f_1(I,J), f_2(I,J))=…
        S_u:  …= A(f_3(I,J), f_4(I,J))
        …
    endfor
endfor
```

Fig. 2 A doubly nested loop program model.

First, we define a program formally as follows.

Definition 1: A sequential program is represented as $P = <\Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta>$, where

- $\Im$ is the set of instructions. An instruction is an indivisible unit such as a simple arithmetic operation on program variables.
- σ_s is the depth, or the number of surrounding loops, of instruction s.
- $\vartheta_s(i)$ is an affine expression derived from the loop bounds such that i is a valid loop index for instruction s, if and only if $\vartheta_s(i) \geq 0$.
- $\wp_{zsr}(i)$ is the affine array index expression in the rth array reference to array z in instruction s.
- ω_{zsr} is true if and only if the rth array reference to array z in instruction s is a write operation.
- γ_{zsr} is true if and only if the rth array reference to array z in instruction s is a read operation.
- $\eta_{ss'}$ is the number of common loops shared by instruction s and s'.

The access patterns in a program define the constraints of program transformations. Informally, there is a data dependence from an access function $\wp$ to another access function $\wp'$, if and only if some instance of $\wp$ uses a location that is subsequently used by $\wp'$, and one of the accesses is a write operation. A data dependence set of a program contains all pairs of data-dependent access functions in the program. Usually, an iteration denoting a series of statements in the loop body is a unit of work assigned to a processor. Therefore, the dependence constraints inside iterations could be ignored. All the dependences discussed in this paper include only cross-iteration dependences. In our program model, as shown in Fig. 2, statement S_d defines elements of array A, and statement S_u uses them. Dependence exists between S_d and S_u whenever both refer to the same element of array A. If S_d defines an element and S_u uses it in a subsequent iteration, there is a cross-iteration flow dependence between S_d and S_u and will be denoted by $(S_d, S_u) \in R^f_\delta$. On the other hand, if S_u uses an element defined by S_d at a later iteration, the dependence is called the cross-iteration anti-dependence and will be denoted by $(S_u, S_d) \in R^a_\delta$.

One of computing data dependence methods is to solve a set of linear diophantine equations formed by iteration boundaries. The data dependence set R contains two pairs of access functions: $<\wp_{b11}, \wp_{b21}>$ and $<\wp_{a11}, \wp_{a21}>$. The constraints are $\vartheta_{su} \geq 0$ and $\vartheta_{sd} \geq 0$, respectively. The loop in Fig. 2 carries cross-iteration dependences if and only if there exists four integers, i_1, j_1, i_2 and j_2, satisfying the system of linear diophantine equations given by Eq. (1) and the system of inequalities given by Eq. (2).

$$f_1(i_1, j_1)=f_3(i_2, j_2) \text{ and } f_2(i_1, j_1)=f_4(i_2, j_2), \text{ where } (i_1, j_1) \text{ and } (i_2, j_2) \in (I, J), \quad (1)$$

$$L_1 \leq i_1, i_2 \leq U_1 \text{ and } L_2 \leq j_1, j_2 \leq U_2. \quad (2)$$

A Dependence Convex Hull (DCH) [18] is a convex polyhedron and a subspace of the solution space. There are two approaches for solving the system of diophantine equations in Eq. (1). One way is to set i_1 to x_1, j_1 to y_1, and then solve i_2 and j_2 respectively. Here, i_1, j_1, i_2, j_2 and its inequalities can be represented as shown in Equation (3), which forms DCH and is denoted by DCH1.

$$(i_1, j_1, i_2, j_2) = (x_1, y_1, g_1(x_1, y_1), g_2(x_1, y_1)), \quad L_1 \leq x_1, g_1(x_1, y_1) \leq U_1 \text{ and } L_2 \leq y_1, g_2(x_1, y_1) \leq U_2 \quad (3)$$

The other way is to set i_2 to x_2, j_2 to y_2 and then solve i_1 and j_1 respectively. Here, i_1, j_1, i_2, j_2 and its inequalities can be represented as shown in Equation (4), which forms DCH and is denoted by DCH2.

$$(i_1, j_1, i_2, j_2) = (g_3(x_2, y_2), g_4(x_2, y_2), x_2, y_2), \quad L_1 \leq g_3(x_2, y_2), x_2 \leq U_1 \text{ and } L_2 \leq g_4(x_2, y_2), y_2 \leq U_2 \quad (4)$$

Clearly, if we have a solution i_1, j_1 in DCH1, we will have a solution i_2, j_2 in DCH2, because each of them is derived from the same set of equations. If iteration (i_2, j_2) is dependent on iteration (i_1, j_1), we will have a dependence vector $D(x, y)$ with $d_i(x, y) = i_2 - i_1$ and $d_j(x, y) = j_2 - j_1$. Therefore, for DCH1, we have

$$d_i(i_1, j_1) = g_1(i_1, j_1) - i_1 \text{ and } d_j(i_1, j_1) = g_2(i_1, j_1) - j_1. \quad (5)$$

For DCH2, we have

$$d_i(i_2, j_2) = i_2 - g_3(i_2, j_2) \text{ and } d_j(i_2, j_2) = j_2 - g_4(i_2, j_2). \quad (6)$$

After introducing some background and related information, we will present the concepts of some our

proposed mechanisms. An arrow represents dependence in an iteration space here. The arrow's head indicates the dependence head, and its tail is known as the dependence tail. The notations of flow-dependence head (tail) set and anti-dependence head (tail) set are denoted by Head(R_δ^f) (Tail(R_δ^f)) and Head(R_δ^a) (Tail(R_δ^a)). Then the unique head (tail) set [6] is a set of integer points in the iteration space that satisfies the following conditions: (1) It is the subset of one of the DCHs (or is the DCH itself); (2) it contains all the dependence arrows' heads (tails), but does not contain any other dependence arrows' tails (heads). We will first examine the concept of DCH1 and DCH2 because it can partition the iteration space into unique sets.

***Lemma 1* [7]:** For a nested loop, DCH1 contains all flow-dependence tails and all anti-dependence heads (if they exist), and DCH2 contains all anti-dependence tails and all flow-dependence heads (if they exist). Thus,

$$\left\{\forall\, i \in \Im^{\sigma s} \,\middle|\, i \in (Tail\ (R_\delta^f) \vee Head\ (R_\delta^a))\right\} \subseteq DCH\ 1 \qquad and$$

$$\left\{\forall\, i \in \Im^{\sigma s} \,\middle|\, i \in (Tail\ (R_\delta^a) \vee Head\ (R_\delta^f))\right\} \subseteq DCH\ 2\,.$$

Lemma 1 tells us that DCH1 and DCH2 may contain more than one unique set and two kinds of unique sets in DCH1 and DCH2 are also given. On the contrary, the following lemma states the conditions for DCH1 and DCH2 to be unique sets.

***Lemma 2* [7]:** For a nested loop, if $d_i(x, y) = 0$ does not pass through any DCH, there will be only one kind of dependence, either flow- or anti-dependence, and DCH itself is the unique head set or the unique tail set.

***Lemma 3* [7]:** For a nested loop, if $d_i(x_1, y_1) = 0$ does not pass through DCH1, then $d_i(x_2, y_2) = 0$ will not pass through DCH2.

***Lemma 4* [7]:** For a nested loop, if $d_i(x_1, y_1) = 0$ ($d_i(x_2, y_2) = 0$) does not pass through DCH1(DCH2), and DCH1 (DCH2) is on the side of $d_i(x_1, y_1) > 0$ ($d_i(x_2, y_2) > 0$), then DCH1 (DCH2) is a flow-dependence unique tail (head) set. Otherwise, if DCH1 (DCH2) is on the side of $d_i(x_1, y_1) < 0$ ($d_i(x_2, y_2) < 0$), then DCH1(DCH2) is an anti-dependence unique head (tail) set.

Now, we have found that if $d_i(x_1, y_1) = 0$ does not pass through DCH1, then both DCH1 and DCH2 are unique sets and the points in them have the same property. DCH1 (DCH2) may contain dependence heads and tails when $d_i(x_1, y_1) = 0$ ($d_i(x_2, y_2) = 0$) passes through it. This makes it difficult finding unique head and tail sets. Lemmas 5 and 6 will show some common attributes when $d_i(x_1, y_1) = 0$ passes through DCH1(DCH2).

***Lemma 5* [7]:** For a nested loop, if $d_i(x, y) = 0$ passes through a DCH, it will divide DCH into a unique tail set and a unique head set. Furthermore, $d_j(x, y) = 0$ determines the inclusion of $d_i(x, y) = 0$ in one of the sets.

***Lemma 6* [7]:** For a nested loop, if $d_i(x_1, y_1) = 0$ passes through DCH1(DCH2), then DCH1(DCH2) is the union of a flow-dependence unique tail (head) set and an anti-dependence unique head (tail) set.

Based on the properties described above, there are various combinations of overlaps of these unique sets. We will illustrate these properties by the following example:

```
for I = 1, 10
    for J = 1, 10
        A (2*J+3,I+J+5) = .....
        ...= A (2*I+J-1,3*I-1)
    endfor
endfor
```

Fig. 3. a doubly nested loop.

The set of inequalities and dependence distances of the loop in Fig. 3 are computed as follows.

DCH1: DCH2: (7)

$$1 \le i_1 \le 10\ and \qquad\qquad 1 \le i_2 \le 10\ and$$

$$1 \le j_1 \le 10\, and \qquad\qquad 1 \le j_2 \le 10\, and$$

$$1 \le \frac{i_1}{3} + \frac{j_1}{3} + 2 \le 10\, and \qquad\qquad 1 \le 2i_2 - \frac{j_2}{2} - 4 \le 10\, and$$

$$1 \le -\frac{2i_1}{3} + \frac{4j_1}{3} \le 10 \qquad\qquad 1 \le i_2 + \frac{j_2}{2} - 2 \le 10$$

$$d_i(i_1, j_1) = -\frac{2i_1}{3} + \frac{j_1}{3} + 2, \qquad\qquad d_i(i_2, j_2) = -i_2 + \frac{j_2}{2} + 4,$$

$$d_j(i_1, j_1) = -\frac{2i_1}{3} + \frac{j_1}{3} \qquad\qquad d_j(i_2, j_2) = -i_2 + \frac{j_2}{2} + 2$$

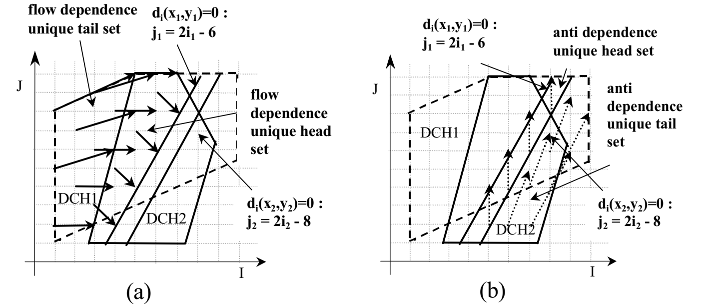

Fig.4 DCHs and the unique head (tail) sets of loop in Fig. 3.

Fig. 4 shows DCHs and the unique head (tail) sets of the loop in Fig. 3. Clearly, $d_i(x_1, y_1) = 0$ divides DCH1 into two areas. The area on the side of $d_i(x_1, y_1) < 0$ is an anti-dependence unique head set, which is on the right side of $d_i(x_1, y_1) = 0$ as shown in Fig. 4(b). Similarly, the area on the side of $d_i(x_1, y_1) > 0$ is a flow-dependence unique tail set, which is on the left side of $d_i(x_1, y_1) = 0$ as shown in Fig. 4 (a). $d_i(x_2, y_2) = 0$ also divides DCH2 into two areas. The area on the side of $d_i(x_2, y_2) < 0$ is an anti-dependence unique tail set, which is on the left side of $d_i(x_2, y_2) = 0$ as shown in Fig. 4 (b). The area on the side of $d_i(x_2, y_2) > 0$ is the flow-dependence unique head set, which is on the right side of $d_i(x_2, y_2) = 0$ as shown in Fig. 4 (a).

Our approach is based on convex hull theory [18]. We use the lines $d_i(i, j) = 0$ and $d_j(i, j) = 0$ to partition the iteration space into four unique sets. All possible sets partitioned by

$d_i(i, j)$ and $d_j(i, j)$ are summarized in Table 1 according to the above lemmas. Table 1(b) shows each set that is a part of a line segment, which is partitioned according to the sign of $d_j(i_1, j_1)$ and $d_j(i_2, j_2)$.

Table 1. (a) The different sets partitioned by $d_i(i, j)$ and $d_j(i,j)$, (b) The case of $d_i(i_1, j_1) = 0$ or $d_i(i_2, j_2) = 0$.

(a)

		$d_i(i_1, j_1)$ > 0	$d_i(i_1, j_1)$ < 0	$d_i(i_1, j_1)$ = 0
$d_i(i_2, j_2)$	> 0	FT, FH DCH_1,DCH_2	FH , AH DCH_2,DCH_1	Refer to (b)
	< 0	FT , AT DCH_1,DCH_2	AT , AH DCH_2,DCH_1	
	= 0	Refer to (b)		

(b)

$d_j(i_1, j_1)$ > 0	$d_j(i_1, j_1)$ < 0	$d_j(i_1, j_1)$ = 0	$d_j(i_2, j_2)$ > 0	$d_j(i_2, j_2)$ < 0	$d_j(i_2, j_2)$ = 0
FT DCH_1	AH DCH_1	No cross-iteration dependence	FH DCH_2	AT DCH_2	No cross-iteration dependence

FT : Flow dependence Tail set
FH : Flow dependence Head set
AT : Anti dependence Tail set
AH : Anti dependence Head set
DCH_1 and DCH_2 are regions they belong to

Here, we use memory space to gain the benefits of parallel execution because the anti-dependence can be avoided by copy renaming. And the extreme points of the convex hulls may have real coordinates. Punyamurtula and Chaudhary [11] proposed an algorithm to convert these extreme points with real coordinates to extreme points with integer coordinates called the Integer Dependence Convex Hull (IDCH). In the following, we will present the concepts of some our proposed mechanisms.

The Parallelization Part Splitting (PPS) [7] is introduced firstly, which splits fully parallelizable part of the iteration space for parallel execution. Because non-uniform dependence loops are irregular, total parallelizable iterations may occupy most of the iteration space. If we can split the parallelizable region out from IDCH in advance using the concept of loop splitting in uniform dependence loop partitioning, and then execute each of them in parallel, it will greatly enhance the speedup of non-uniform dependence loops. Fig. 5(a) and Fig. 5(b) show the Left-tile and the Right-tile respectively. We can first execute iterations on the Left-tile in parallel, and then execute iterations that are tiled using ordinary parallelization or our following mechanisms. Finally, iterations on the Right-tile can be executed in parallel. Due to elimination of all unnecessary dependencies in both Left-tile and Right-tile, performance of the PPS is not constrained by the minimum dependence distance as the conventional methods behaved.

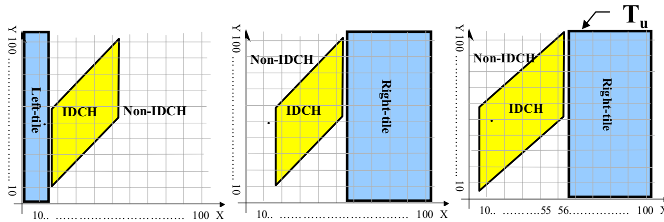

Fig. 5. Tiling of the PPS mechanism: (a) tiling of the left tile, (b) tiling of the Right-tile, (c) tiling for Model 1 at Table 2.

Second, the Partial Parallelization Decomposition (PPD) [7] mechanism is presented to partition the iteration space into different parts and handles them by conventional techniques. It avoids the loop parallelism restricted by the minimum dependence distance in the whole iteration space. If an iteration space could be partitioned into parts, the minimum dependence distance of each part may be larger than the original minimum dependence distance. This decomposition method could exploit more parallelism degree in each part of the iteration space. However, the minimum dependence distance may be larger if iterations are tiled along with other loop index, henceforth, loop interchange could be applied. Fig. 6 shows an example of applying PPD.

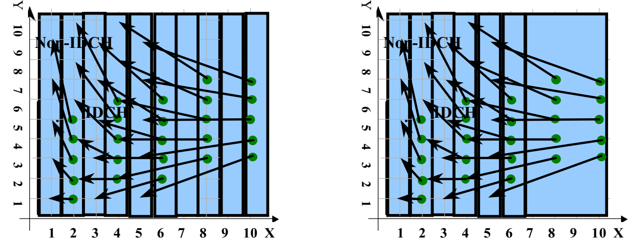

Fig. 6. (a) Tile before PPD, (b) tile after PPD.

Third, because loop interchange may extract parallelism in other loop dimensions, the Irregular Loop Interchange (ILI) [7] scheme is proposed to detect whether the transformation is legal or not. Fig. 7 indicates an example of applying ILI mechanism for Model 4 at Table 2.

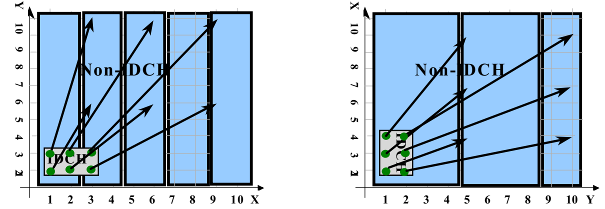

Fig. 7. (a) Tile before ILI, (b) tile after ILI.

Finally, the Growing Pattern Detection (GPD) [7] technique detects the dependence vector function whether it is increasing or decreasing progressively, and then tiles the non-uniform dependence loop according to this dependence vector function.

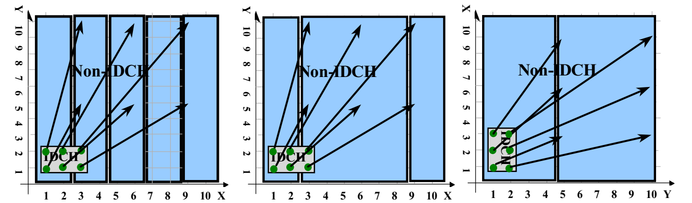

Fig. 8. (a) Original Tiling, (b) tiling with GPD and (c) tiling with ILI and GPD.

Tiling after the GPD mechanism is shown in Fig. 8 (b) for the model 4 at Table 2; and the parallelism is greater than that yielded by the original tiling method, as shown in Fig. 8 (a). The dependence index J is also a growing pattern, so we use the ILI and GPD methods. After applying these two mechanisms, the parallelism degree is further improved, as

displayed in Fig. 8 (c).

After introducing our proposed mechanisms, the categorization of non-uniform dependence loops is presented in the following section.

3. CATEGORIZATION OF NON-UNIFORM DEPENDENCE LOOPS

We have proposed several effective partition mechanisms for non-uniform dependences loops with different properties. Then, a systematical scheme is presented here to classify non-uniform dependence loops and partition them with suitable partitioning schemes. At first, the non-uniform dependence loops are classified into four types, (1) Non-uniform dependence loops with special dependence vector patterns, (2) non-uniform dependence loops without Inherent Serial Area, (3) non-uniform dependence loops with Inherent Serial Area which does not cover all of the iteration space and (4) non-uniform dependence loops with Inherent Serial Area covering all of the iteration space.

In the following, the mechanisms for partitioning each type of non-uniform dependence loops will be proposed.

Type 1: Non-uniform dependence loops with special dependence vector patterns.

If the loop's dependence vector contains special patterns which could be partitioned easily, we could detect and partition them based on their dependence vector patterns. The growing pattern detection (GPD) mechanism could be applied here to detect special dependence vector patterns and parallelize them. When the irregular loop interchange (ILI) mechanism detects that loop interchange is legal, the loop could be interchanged to exploit more parallelism from other dimensions.

Type 2: Non-uniform dependence loops without Inherent Serial Area.

This type of non-uniform dependence loops could be parallelized into partitions in which iterations could be executed in parallel. Our optimized three region partitioning (OTRP) [6, 10] mechanism is suitable for such kind loops. At first, the iteration space is partitioned into three regions $Area_1$, $Area_2$, and $Area_3$. The iterations in $Area_1$ and $Area_2$ could be executed in parallel. The iterations in $Area_3$ have to be executed in serial. The following is the concept of partitioning $Area_1$, $Area_2$, and $Area_3$:

1) $Area_1$: This region may include anti-dependence heads and flow dependence tails, but should not include flow dependence heads. As shown in Table 1, the case of $d_i(i_2, j_2) < 0$ and ($d_i(i_2, j_2) = 0$ and $d_j(i_2, j_2) < 0$) are included. On the other hand, the area in the case of ($d_i(i_1, j_1) > 0$ and $d_i(i_2, j_2) > 0$) subtracting DCH2 is also included. The definition of $Area_1$ is given: $Area_1 = Area_{11} \cup Area_{12} \cup Area_{13}$, where $Area_{11} = \{(i_2, j_2) \mid d_i(i_2, j_2) < 0\}$, $Area_{12} = \{(i_2, j_2) \mid d_i(i_2, j_2) = 0$ and $d_j(i_2, j_2) < 0\}$, and $Area_{13} = \{(i_1, j_1) \mid d_i(i_1, j_1) > 0\} \cap \{(i_2, j_2) \mid d_i(i_2, j_2) > 0\} - \{\{(i_1, j_1) \mid d_i(i_1, j_1) > 0\} \cap \{(i_2, j_2) \mid d_i(i_2, j_2) > 0\} \cap DCH2\}$.

Because there may be anti-dependence heads and tails in different areas, copy-renaming is applied to remove anti-dependences. And then the iterations in $Area_1$ can be fully executed in parallel [6, 10].

2) $Area_2$: In this region, the flow dependence and anti-dependence heads should be included. As shown in Table 1, the cases of ($d_i(i_1, j_1) < 0$ and $d_i(i_2, j_2) > 0$), ($d_i(i_1, j_1) = 0$ and $d_j(i_1, j_1) < 0$) and ($d_i(i_2, j_2) = 0$ and $d_j(i_2, j_2) > 0$) are included. The definition of $Area_2$ is given: $Area_2 = Area_{21} \cup Area_{22} \cup Area_{23} \cup Area_{24}$, where $Area_{21} = \{(i_1, j_1) \mid d_i(i_1, j_1) < 0\} \cap \{(i_2, j_2) \mid d_i(i_2, j_2) > 0\}$, $Area_{22} = \{(i_1, j_1) \mid d_i(i_1, j_1) = 0$ and $d_j(i_1, j_1) < 0\}$, $Area_{23} = \{(i_2, j_2) \mid d_i(i_2, j_2) = 0$ and $d_j(i_2, j_2) > 0\}$, and $Area_{24} = \{\{(i_1, j_1) \mid d_i(i_1, j_1) > 0\} \cap \{(i_2, j_2) \mid d_i(i_2, j_2) > 0\} \cap DCH2\} - \{\{(i_1, j_1) \mid d_i(i_1, j_1) > 0\} \cap \{(i_2, j_2) \mid d_i(i_2, j_2) > 0\} \cap DCH1 \cap DCH2\}$.

This $Area_2$ contains anti-dependence heads and flow dependence heads whose corresponding tails exist in Area. Once $Area_1$ and $Area_3$ are executed, the iterations in $Area_2$ can be fully executed in parallel.

3) $Area_3$: This region is the rest of the iteration space excluding $Area_1$ and $Area_2$. In this region, flow dependence heads and tails are included. As shown in Table 1, the cases of (($d_i(i_1, j_1) > 0$ and $d_i(i_2, j_2) > 0$) $\cap$ DCH1 $\cap$ DCH2) and ($d_i(i_1, j_1) = 0$ and $d_j(i_1, j_1) > 0$) are included. The definition of $Area_3$ is given: $Area_3 = Area_{31} \cup Area_{32}$, where $Area_{31} = \{(i_1, j_1) \mid d_i(i_1, j_1) > 0\} \cap \{(i_2, j_2) \mid d_i(i_2, j_2) > 0\} \cap DCH1 \cap DCH2$ and $Area_{32} = \{(i_1, j_1) \mid d_i(i_1, j_1) = 0$ and $d_j(i_1, j_1) > 0\}$.

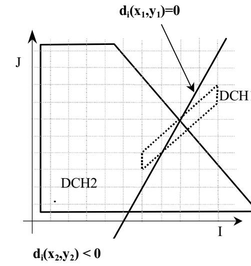

Fig. 9. The non-uniform dependence loop of model 1 in Table 2 with dependence of type 2.

Because $Area_1$ contains only flow dependence tail sets after copy-renaming and $Area_2$ contains only flow dependence head sets after copy-renaming, they could be executed in parallel respectively. However, $Area_3$ contains both flow dependence head and tail sets. The iterations in it must be executed in serial, i.e. $Area_3$ is an inherently serial region. Due to type 2 doesn't have ISA part, the execution of the partitioning follows the sequence of $Area_1 \rightarrow Area_2$. Iterations in $Area_1$ and $Area_2$ can be executed in parallel. The complexity of the algorithm is bounded by the complexity of dependence convex hull's algorithm. Thus, it is effective and can be constructed in current parallel compilation environments. We will give an example for the program model 1 in Table 2 with $U_I = U_J = 10$, the iteration space contains only anti-dependence and can be eliminated by copy renaming. The ISA region does not exist in the

iteration space as shown in Fig. 9.

Type 3: Non-uniform dependence loops with Inherent Serial Area which does not cover all of the iteration space.

This type loops still have iterations which could be executed in parallel, so we could partition them in advance by PPS and the first stage of our TSP [9] mechanism. After applying above mechanisms, the iterations which could not be executed in parallel, i.e. the ISA area, could be further partitioned into parallel partitions by the TSP's second stage and PPD mechanism.

Our two stage partitioning (TSP) [9] mechanism is based on the three-region partitioning technique and the dependence convex theory to reduce the complexity of our algorithm. TSP could divide the iteration spaces into three regions, $Area_1$, $Area_2$, and $Area_3$ in the first stage. The iterations in the regions $Area_1$ and $Area_2$ can be executed in parallel and $Area_3$ must be handled furthermore in the second stage. The definitions of $Area_1$, $Area_2$, and *Area* are similar to that of OTRP mechanism, and the execution order is $Area_1 \rightarrow Area_3 \rightarrow Area_2$. Iterations in $Area_1$ and $Area_2$ can be executed in parallel, $Area_3$ cannot be executed in parallel, and further processes are needed in the second stage.

The main idea of the second stage of the TSP mechanism is as follows. If we could find the flow dependence head iteration, *(x, y),* that occurs first (i.e., its execution order is lexicographical firstly) in the range of the loop given by $Area_3$ and its corresponding tail is inside the $Area_3$. Due to there is no dependences between these iterations, all the iterations, *(i, j)*, in the range of these iterations whose lexicographical order is less than the iteration *(i, j)* and inside the $Area_3$ could be executed in parallel. Hence, we could make the iterations into a partition. The range of the iterations whose lexicographical order is the same and greater than the iteration *(i, j)* and inside the $Area_3$ could then be partitioned in the same way. We could repeat this procedure for the rest of the loop iterations inside the $Area_3$. However, if the tail of the dependence head is located at one of the previous partitions, there is no dependence. Thus, we could ignore the dependence heads explored above when we partition these extra iterations.

Fig. 10 illustrates the result of applying the first stage of TSP method to the loop of Fig. 3 in which the iteration space contains non-empty Inherent Serial Region. $Area_1$ is the region excluding the relation $(((J > 2I-8) \cap (J<2I-6)) \cup ((2I-8 < J) \cap (J<4-18I) \cap (J<-2I+16)))$, $Area_3$ is $((J > 2I-8) \cap (J<4I-28))$, and $Area_2$ is $(((J < 2I - 6) \cap ((2I-8 < J)) \cup ((J>4I-28) \cap (J<(3+2I)/4) \cap (J>2I-6)))$.

To partition the iteration space with non-empty ISA, we should apply the first stage of TSP or the PPS mechanism in advance. In this case, we find that we could exploit more parallelism after applying the first stage of the TSP mechanism, and the ISA region could be further partitioned by the second stage of the TSP mechanism.

Type 4: Non-uniform dependence loops with Inherent Serial Area covering all of the iteration space.

To partition the iteration space with the ISA covering all of it, we could apply ODCHP [8-9] or PPD mechanisms. All the iteration space could be partitioned into several regions in which iterations can be executed in parallel. We could choose one of the ODCHP and PPD mechanisms based on which has the smaller number of partitions.

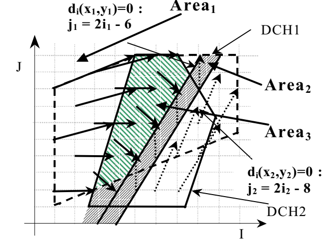

Fig. 10. Regions of the loop partitioned by the first stage of TSP for Fig. 3.

The concept of ODCHP is described as follows. If we could find the dependence head iteration, *(x, y),* that occurs first (i.e., its execution order is lexicographically the first) in the range of the loop given by $l_1 \le x \le u_1$ and $l_2 \le y \le u_2$, then all the iterations, *(i, j)*, in the range of ($l_1 \le i \le x\text{-}1$ and $l_2 \le j \le u_2$) plus ($i = x$ and $l_2 \le j \le y\text{-}1$) could be executed in parallel since there are no dependences between these iterations. Thus, we could make the iterations in the range of ($l_1 \le i \le x\text{-}1$ and $l_2 \le j \le u_2$) plus ($i = x$ and $l_2 \le j \le y\text{-}1$) into a partition with size $(x\text{-}1_2)(u_2\text{-}l_2\text{+}1) + (y\text{-}l_2)$. The range of ($x \le i \le u_1$ and $l_2 \le j \le u_2$) plus ($i = x$ and $y \le j \le u_2$) could then be partitioned in the same way. The procedure is repeated for the rest iterations. However, there is no dependence if the tail of the dependence head is located at one of the previous partitions. The definition of Optimized Dependence Convex Hull Partition (ODCHP) is given as follows. For a nested loop, the k_{th} partition of the ODCHP method, $ODCHP_k$, begins from the end of the next iteration of the previous partition to the previous iteration of the first head node with a tail node that belongs to the current partition.

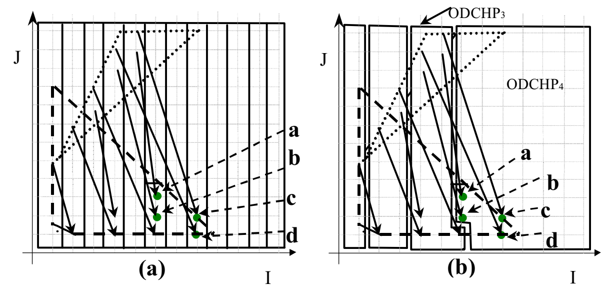

Fig. 11 (a) An example partitioned by the Minimum dependence distance tiling without inclusion property, (b) an example of ODCHP partitioning with the inclusion proerty.

By using the Inclusion Property [8-9], for a nested loop, if all the corresponding dependence tails of dependence heads belong to the previous ODCHP partitions, then the iterations from the source node of the current ODCHP partition to the node of the last dependence head

with lexicographical execution order can be executed in parallel.

As shown in Fig. 11 (a), the iteration space in Fig. 1 (a) is partitioned by the minimum dependence distance tiling method. Even the corresponding dependence tails of dependence heads a, b, c, d belong to the previous tiles, they are also tiled with less dependence distance than the ODCHP scheme. We find that the number of iterations executed in parallel increases greatly by using ODCHP. As shown in Fig. 11 (b), all the corresponding dependence tails of dependence heads a, b, c, d belongs to the previous ODCHP partition, $ODCHP_3$. Thus, the iterations from the source node of the $ODCHP_4$ partition to the node of the last dependence head c with lexicographically execution order can be executed in parallel. For a nested loop, the iterations inside the same ODCHP partition can be executed in parallel, and the number of iterations inside each partition tiled by the ODCHP mechanism is a greedy maximum under the constraint of lexicographical execution order. By using an improved integer programming technique, we could obtain the generalized and optimized algorithm to exploit more parallelism in nested loops with non-uniform dependences.

According to the partition methods discussed above, we propose a complete non-uniform partitioning algorithm in the following. They are classified based on the concept of dependence convex hulls and dependence vector lines.

```
Algorithm Categorization;
/* Input: The nested loop L with non-uniform dependences
   Detection Mechanisms: GPx ,GPy, ILI, ISA and IS
   Partitioning Mechanisms: GPD, ILI, PPS, OTRP, ODCHP, PPD
                            and TSP.
  Output: The nested loop L1 after partitioning */
Begin
If (GPx (L)== True) then
                         /*Axis x is detected as growing pattern*/
     L1=GPD(L);     /*Growing Pattern Detection Mechanism*/
Else if (GPy (L)== True) and (is(ILI(L)) == true) then
                         /*Axis y contains growing Pattern*/
     L1=ILI(L);       /*Irregular Loop Interchange Mechanism*/
     L1=GPD(L);     /*Y axis is applied GPD Mechanism*/
Else if (Sizeof(ISA(L)) < Sizeof(IS(L)))
                   /*Inherit Serial Region is less than Iteration space*/
     If (Parallel(ISA(L)) == True) /*OTRP is applied if the     */
          L1=OTRP(L);             /* inherit serial region can  */
                                  /*be executed in parallel     */
     Else        /*TSP is applied if the inherit serial region can*/
          L1=TSP(L);             /* not be executed in parallel*/
Else if ((Sizeof(ISA(L)) = Sizeof(IS(L))) then
                        /* ISA is equal to the size of iteration space*/
     L1=PPS & ILI(L);       /* apply PPS and ILI mechanisms*/
     If (# of Partitions of (PPD(L)) < # of Partitions of
        (ODCHP(L)))
          L1=PPD(L);   /*PPD is used if better than ODCHP*/
     Else
          L1=ODCHP(L);                    /*ODCHP is used*/
End.
```

In this algorithm, we firstly detect the special pattern of Type 1 discussed above because it can exploit parallelism step by step and its parallelism exploited is very large. We will detect this special pattern in every dimension of loops if combined with the ILI mechanism. If the GPD pattern is detected in the loop, we will partition them in the specific dimension with the maximum GPD parallelism of the specific dimension. Second, we will partition them due to the size of their ISA region. When the size of the ISA region is empty, then the OTRP mechanism can partition the iteration space into two parallel regions. Third, if the ISA region is not empty and it does not cover all of the iteration space, we can apply the TSP mechanism. Finally, if the ISA region covers all of the iteration space, the PPD and ODCHP schemes can be applied and effectively exploited parallelism. The time complexities of the proposed mechanisms are depicted in [6-10], and the time complexity of the categorizing step is bounded by the complexity of dependence convex hull algorithm. By using existing dependence convex hull functions in current parallel compilers, our mechanism is reasonable to be constructed in current parallel compilation environments.

We could parallelize general non-uniform dependence loops without any constraint. In order to show the performance of our mechanisms, we have implemented them in a parallel compilation environment and evaluated them in supercomputer and evaluation environment with large number of processors. We will discuss them in the following section.

4. PERFORMANCE EVALUATIONS

We have constructed our mechanisms in the **SUIF** [20] parallel compilation environment and then evaluated the program models on a CONVEX SPP-1000 system with 8 processors and on a simulator named SEESMA (A Simulation and Evaluation Environment for Shared-Memory Multiprocessor Architecture [14]) which is enhanced from MINT [19]. The evaluated program models are four popular models and two real program kernel code segments [4, 15, 16-18] as shown in Table 2 and 3.

We used the model 4 in Table 2 to evaluate different mechanisms. Fig.12 shows the speedup of our techniques versus minimum dependence distance method and uniformization technique. For this model, our technique got a better performance. The loop bounds of this model are 30. Thus, iterations of this model are much larger than the number of processors in the IBM SPP1000 system.

Fig. 13 shows the speedup of our technique OTRP versus ITRP combined with MDT (Minimum Dependence distance Tiling) and VSP (Variable Size Partitioning) techniques. Our technique delivers a better performance while the loop bounds of this example are set to 100 as shown in Fig. 13 (a). As the number of processor increases, the performance of the OTRP combined with the VSP

mechanism shows even better than the others. However, the speedup of different mechanisms is nearly linear as shown in Fig. 13 (b) due to the massive parallelism degree and the limited number of processors in the CONVEX SPP-1000 while the loop bound is set to 1000.

Table 2 The standard models.

Model 1	Model 2

for I=1,N **do**
for J=1,M **do**
s1: A(2I,2J)= …
s2: … = A(J+10, I+J+6);
enddo
enddo
	for I=1,N **do**
	for J=1,M **do**
	s1: A(I+J, 3I+J+3) = …
	s2: … = A(I+J+1, I+2J+4);
	enddo
	enddo

Model 3	**Model 4**

for I=1,N **do**
for J=1,M **do**
s1: A(2J+3,I+1) = …
s2: …=A(I+J+3, 2I+1);
enddo
enddo
	For I=1,N **do**
	for J=1,M **do**
	s1: A(3I, 5J) = …
	s2: …… = A(I, J);
	enddo
	enddo

Table 3 The practical code Segments.

Propogate Code Segment	**Swap** Code Segment
DO I=1,Q **DO** J=1,R AR (I, J) = AR(1, J) **CONTINUE** **CONTINUE**	**DO** I=1, 10 **DO** J=1, 10 Y(J, I) = Y(J, N+1-I) **CONTINUE** **CONTINUE**

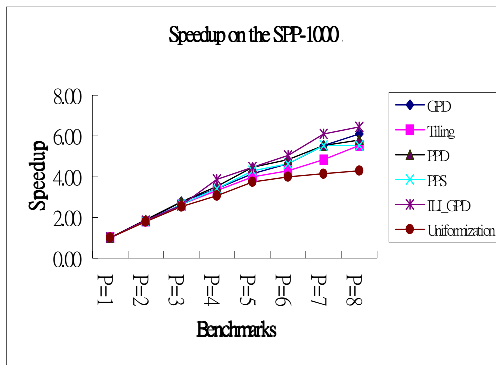

Fig.12. Evaluation on the real machine for model 4.

Fig. 14 shows the speedup comparisons in the SEESMA. For Propogate code segment, our mechanism finds that the flow dependence tail only covers a small region of the iteration space. And after the first partitioning step, all the iteration space is fully parallelized. Thus, in this program code segment, our mechanism shows dramatic better performance than other mechanisms as shown in Fig. 14 (a). For the Swap code segment, because the ITRP, ITRP-MDT and our ODCHP mechanisms partition the iteration space into two parallel regions, their performance is nearly the same and all effective as shown in Fig. 14 (b).

Fig. 15 shows the speedup comparisons in the SEESMA environment. Our technique delivers a better performance while the loop bounds of this example are set to

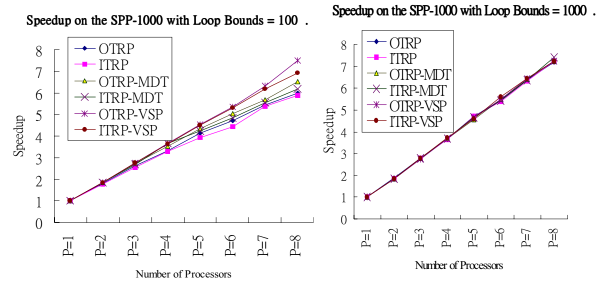

Fig. 13. Evaluation on the real machine for the program in Fig. 3, (a) Speedup on the SPP-1000 with loop bound = 100, (b) loop bound = 1000

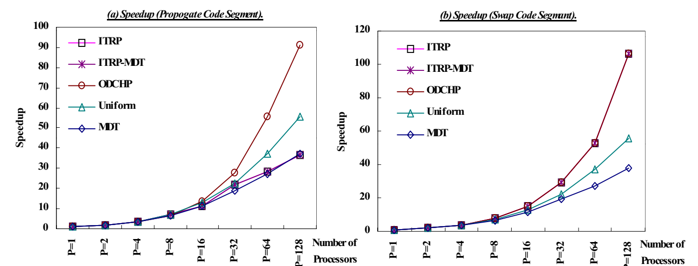

Fig. 14. Evaluation on the SEESMA environment for (a) Propogate, (b) Swap code.

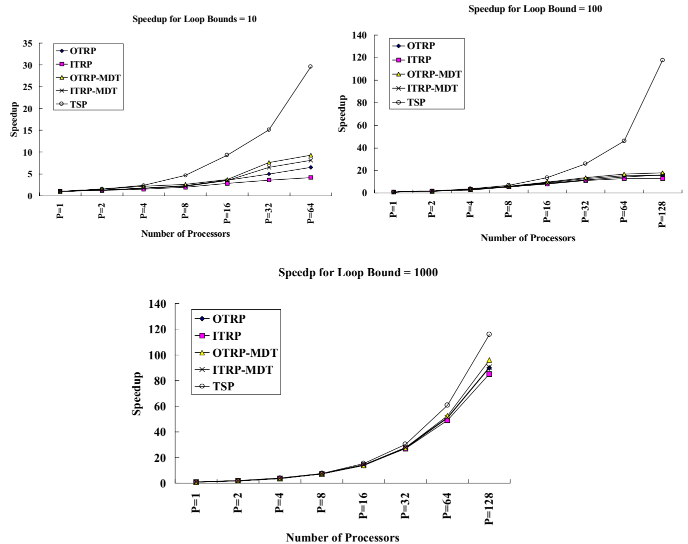

Fig. 15. Speedup on SEESMA for the program in Fig. 3 with (a) loop bounds = 10, (b) = 100, (c) = 1000.

10, 100 and 1000 as shown in Fig. 15 (a), (b) and (c). As the number of processor increases to 128, the performance of the TSP shows even better than the other mechanisms. They are close to the ideal speedup 8.3, 10 and 33 respectively than the speedup running on the CONVEX SPP-1000 which are 8.13, 9.29 and 29.55 as shown in Fig. 15 (a) because of more number of processors can exploit more parallelism degree. It is clear that the TSP exploits more parallelism than either three-region partitioning or unique set oriented partitioning method. In addition, the TSP scheme extracts much more parallelism than the ITRP and OTRP schemes

and other mechanisms do.

5. CONCLUDING REMRKS

In summary, we propose a systematical and effective partitioning mechanism to exploit parallelism from nested loops with non-uniform dependences. As discussed above, we not only classify benchmarks into several specific types but we also apply appropriate schemes according to their attributes. On the other hand, we proof the effectiveness of our categorizations. In the preliminary evaluations, we find that our categorization can partition effectively. Our scheme could extract parallelism from all types of non-uniform dependences. Many aggressive methods may be possible only for specific types of dependence vector patterns. However, our method is the most complete and general one currently. So far, our mechanisms have been implemented in SUIF to be used as a good platform for parallel compilation of non-uniform loop structures.

6. REFERENCES

[1] Banerjee. U., 1988. Dependence Analysis for Supercomputing, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, MA 02061, Boston.

[2] Chen, D.K. and Yew, P.C., 1996. On Effective Execution of Nonuniform DOACROSS Loop, IEEE Transactions on Parallel and Distributed Systems. 7. 5, 463-476.

[3] Cho, C. K. and Lee, M.H., 1997. A Loop Parallelization Method for Nested Loops with Non-uniform Dependence. Int. Conference on Parallel Processing, Dietz, Hank & et al., (Eds), Los Alamitos, CA, 314-321.

[4] Dongarra, J. J., Moler, C. B., Bunch, J. R. and Stewart G. W., 1979. LINPACK Users' Guide by SIAM, Philadelphia.

[5] Ju, J. and Chaudhary, V., 1996. Unique Sets Oriented Partitioning of Nested Loops with Non-uniform Dependences, Int. Conference on Parallel Processing, III, Los Alamitos, California, K. Pingali. (Ed), 45-52.

[6] Der-Lin Pean and Cheng Chen, “An Optimized Three Region Partitioning Technique to Maximize Parallelism of Nested Loops with Non-uniform Dependence,” accepted and to appear in the *Journal of Information Science and Engineering*.

[7] Der-Lin Pean and Cheng Chen, “Effective Parallelization Techniques for Loop Nests with Non-uniform Dependences,” accepted and to appear in the *Journal of Parallel Algorithm and Applications*.

[8] Der-Lin Pean and Cheng Chen, “ODCHP: A New Effective Mechanism to Maximize Parallelism of Nested Loops with Non-uniform Dependences,” accepted and to appear in the *Journal of Systems and Software*.

[9] Der-Lin Pean, Guan-Joe Lai and Cheng Chen, “An Optimized Dependence Convex Hull Partitioning Technique to Maximize Parallelism of Nested Loops with Non-uniform Dependences”, accepted by *Proceeding of the 2000 Int. Conf. on Para. and Distributed Systems, ICPADS'2000*, Iwate, Japan, July 4-7, 2000.

[10] Der-Lin Pean and Cheng Chen, “An Optimized Loop Partition Technique for Maximize Parallelism of Nested Loops with Non-uniform Dependences,” in *Proceeding of the Fifth Workshop on Compiler Techniques for High-Performance Computing*, Chiayi, Taiwan, R.O.C., March 18-19, 1999, pp. 158-171.

[11] Punyamurtula, S. and Chaudhary, V., 1994. Minimum Dependence Distance Tiling of Nested Loops with Non-uniform Dependences, Proc. 6th IEEE Symposium on Parallel and Distributed Processing, 74-81.

[12] Punyanurtula, S., Ju, J., Chaudhary, V. and Roy, S., 1997. Compile Time Partitioning of Nested Loop Iteration Spaces with Non-uniform Dependences, Parallel Algorithms and Applications. 12, 113-141.

[13] Shen, Z., Li, Z. and Yew, P.C., 1989. An Empirical Study on Array Subscripts and Data Dependencies, Int. Conf. on Parallel Processing, Pennsylvania State University, Ris, Fred & Kogge, Peter M. (Ed.), 145-152.

[14] Su, J.P., Wu, C.C. and Chen, C., 1996. Reducing the Overhead of Migratory-Shared Access for the Linked-Based Directory Coherent Protocols in Shared Memory Multiprocessor Systems. Proc. ICS’96 on Computer Architecture, Kaohsiung, Taiwan, R.O.C., Wen-Shyong and Lionel M. Ni (Eds.), 160-167.

[15] Swarztrauber, P. A. and Sweet, R. A., 1979. Efficient Fortran Subprograms for the Solution of Separable Elliptic Partial Differential Equations. ACM Transactions on Mathematical Software, 5. 3, 352-364.

[16] Tseng, S.Y., King, C.T. and Tang, C.Y., 1992. Minimum Dependence Vector Set: A New Compiler Technique for Enhancing Loop Parallelism, Int. Conference on Parallel and Distributed Systems, HsinChu, Taiwan, National Tsing Hua University, 340-346.

[17] Tseng, S.Y., King, C.T. and Tang, C.Y., 1996. Profiling Dependence Vectors for Loop Parallelization, Proc. International Parallel Processing Symposium, Los Alamitos,Cali, 23-27.

[18] Tzen, T. H. and Ni, L.M., 1993. Dependence Uniformization: A Loop Parallelization Technique, IEEE Trans. on Parallel and Distributed System. 4, 547-558.

[19] Veenstra, J.E. and Fowler, R.J., 1994. MINT Tutorial and User Manual, Technical Report, 452, University of Rochester, New York.

[20] Wilson, R., Lam, M.S. and Hennessy, J., 1996. An Overview of the SUIF Compiler System, Computer Systems Lab, Stanford University.

[21] Zaafrani, A. and Ito, M., 1994. Parallel Region Execution of loops with Irregular Dependences. Proc. the Int. Conf. on Parallel Processing, Aug. 15-19, 11-19.