JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **35,** 123–132 (1996)
ARTICLE NO. 0075

Tiling Nested Loops into Maximal Rectangular Blocks

YEONG-SHENG CHEN,* SHENG-DE WANG,† AND CHIEN-MIN WANG‡

**Department of Electronics Engineering, Hwa-Hsia College of Technology, Taipei 235, Taiwan, Republic of China; †Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan, Republic of China; and ‡Institute of Information Science, Academia Sinica, Taipei 115, Taiwan, Republic of China*

In this paper, an approach to tiling nested loops for maximizing parallelism is proposed. The proposed method aims at aggregating independent computations of a loop nest into rectangular blocks and maximizing the block sizes for maximizing parallelism. At first, all the independent computations that can be executed in the first time unit are identified. These computations are called the initially independent computations. Then it is shown that all of them can be collected as a union of rectangular blocks. So, based on these, the entire iteration space of the loops is partitioned into rectangular blocks for maximizing parallelism. The proposed method is formulated as systematic procedures which can easily be implemented in a parallelizing compiler. It is shown that when the wavefront transformation is combined with the proposed method, the loops can always be tiled so that the tile size is greater than one. In comparison with previous work on tiling, the proposed method is shown to have several advantages as summarized in the conclusions of this paper. © 1996 Academic Press, Inc.

I. INTRODUCTION

It is widely recognized that loops provide the largest source of parallelism in common numerical programs. A great deal of research has gone into the development of various techniques for exploiting parallelism within nested loops [3–11, 13–18, 20–27]. Some well-known techniques are wavefront execution [11], hyperplane partitioning [5], time transformation [21], affine scheduling [3], partitioning vector approach [22], minimum distance [14], and loop labeling [4]. Although these methods are useful in maximizing parallelism, they do not aggregate independent computations into regions with fixed sizes and regular shapes. So the compiler cannot easily generate the parallel codes or the parallel codes cannot be efficiently executed due to data reference locality and processor load balance problems.

Much research has shown that the above problems can be relieved by partitioning loops into blocks (tiles) [6, 8, 23, 25]. Tiles provide data reference locality for memory hierarchy utilization; and the compiler can easily generate parallel codes for tiles so that processors are load balanced when executing the codes. So partitioning loops into blocks is a very important optimization issue. Some well-known techniques are described as follows. Wolfe discusses the techniques of strip mining and iteration space tiling [25], which organize the computations in the original loops into chunks of equal size to take advantage of vector registers, caches, or local memory. Nicolau [13] proposes a method called loop quantization to partition nested loops. King and Li [9] discuss the grouping of loop iterations for parallel execution on multicomputers. The cycle shrinking method proposed by Polychronopoulos [16] uses the data dependence graphs of loops to determine which loops can be executed in parallel. Irigoin and Triolet [8] present a method called supernode partitioning, which formulates a general condition for determining the admissible tiles formed by multiple hyperplanes. Schreiber and Dongarra [20] discuss a heuristic method of choosing a subset of dependence vectors for tiling loops. Ramanujam and Sadayappan [18] formulate an approach to determine the hyperplanes and the tile sizes for reducing the communication cost in distributed memory systems. As for the code generation issue for loop tiling, Ancourt and Irigoin [1] discuss how to generate codes for the tiles defined by a set of linear equations, and the code generation techniques for nonunimodular loop transformations [12, 17] can also provide a solution to this problem.

In this paper, we propose a new approach to tiling nested loops for exploiting parallelism. The proposed approach aims at aggregating as many independent computations as possible into a tile in order to maximize parallelism. At first, we describe a systematic procedure to find *all* the computations that are independent when the loops are to be executed. All these independent computations are collected together as a union of sets, which are called initially independent computation sets. Then, we show that *all* the initially independent computations can be aggregated into rectangular blocks. So, based on these, the original loops can be partitioned into rectangular blocks to maximize parallelism. Since each partitioned region is block-shaped, the code generation is very easy. Also, we show that if the wavefront transformation is combined with the proposed method, the original loops can always be tiled so that the tile size is greater than one.

0743-7315/96 $18.00
Copyright © 1996 by Academic Press, Inc.
All rights of reproduction in any form reserved.

II. TERMINOLOGY AND MOTIVATION

A. Terminology

The program model considered in this paper is the uniform dependence loop nests [1, 22, 24] whose dependence pattern is the same for each iteration of loops. A loop iteration is considered as the basic scheduling unit and is called a *computation*. A uniform dependence loop nest of depth n is denoted as $L_n(V, D)$, where V is the set of indexed points each of which represents one loop iteration, and D is the dependence matrix in which each column is a dependence vector [26] that describes the interiteration dependence of the loops. Without loss of generality, the loop nest is assumed to have been normalized so that for each loop level i, $1 \le i \le n$, the lower bound of loop index is 0 and the loop index increment is 1. The interiteration dependences of a loop nest $L_n(V, D)$ can be represented by an Iteration Space Dependence Graph (ISDG) [26], in which a directed edge represents a data dependence relation of two computations.

B. Motivation

The dependence relations will be changed due to the shape and size of the index set. Shang and Fortes [22] introduce the concept of *pseudo-dependences* to characterize dependence relations that do not consider both the boundary conditions of the iteration space and the direction of the dependence vectors. Two computations $\mathbf{v}_1$ and $\mathbf{v}_2$ are pseudo-dependent on each other if there is a vector $\boldsymbol{\lambda} \in Z^m$ such that $\mathbf{v}_1 = \mathbf{v}_2 + D\boldsymbol{\lambda}$ (m is the number of dependence vectors).

Most existing techniques for exploiting parallelism within loops are based on pseudo-dependences. However, in reality, pseudo-dependences are not equivalent to "real" dependences. A computation $\mathbf{v}_1$ may not depend on a computation $\mathbf{v}_2$ even if $\mathbf{v}_1 = \mathbf{v}_2 + D\boldsymbol{\lambda}$, because the dependence path may exit the index space [2].

We observe that when loops are to be executed, the computations whose all real dependences are satisfied can be easily identified. These computations are called *initially independent computations*. Intuitively, to exploit the optimal (maximal) parallelism within a loop nest is to identify *all* the computations that can be executed in the first time unit, and then *all* those can be executed in the second time unit, ..., and so on. So identifying all the initially independent computations is a good starting point for exploiting maximal parallelism. Motivated by this observation, we propose a method of finding all the initially independent computations, and then, based on these, of tiling the loops for maximizing parallelism.

III. BASIC STEPS OF THE PROPOSED APPROACH

A. Some Definitions

For a loop nest $L_n(V, D)$, R is called a *computation set* if $R \subset V$.

DEFINITION 1. *Independent computations, independent computation sets*: A computation is *independent* iff all its dependence sources either have completed their execution or are outside the iteration space. Note that when we talk about an independent computation, we imply it is executable at a certain time unit. A computation set R is called an *independent computation set* (ICS) iff, $\forall \mathbf{v} \in R$, $\mathbf{v}$ is an independent computation.

DEFINITION 2. *Initially independent computations, initially independent computation sets*: A computation is *initially independent* iff all its dependences have been satisfied when the loops are to be executed. An *initially independent computation set* (IICS) is an independent computation set in which every computation is initially independent.

An initially independent computation set is *maximal* iff it aggregates *all* the initially independent computations.

DEFINITION 3. *Boundary block computation sets*: A *boundary block computation set* $R = [r_1, r_2, ..., r_n]$ of a loop nest L_n is a set of *initially independent* computations $(p_1, p_2, ..., p_n)^{\mathrm{T}}$ where

(a) $\forall i$, $1 \le i \le n$, $-u_i - 1 \le r_i \le u_i + 1$ and $r_i \in Z$ (u_i denotes the upper bound of the ith loop);
(b) $\forall i$, $1 \le i \le n$, if $r_i < 0$ then $u_i + r_i + 1 \le p_i \le u_i$;
(c) $\forall i$, $1 \le i \le n$, if $r_i > 0$ then $0 \le p_i \le r_i - 1$; and
(d) if $\exists i, 1 \le i \le n$, such that $r_i = 0$, then R is an empty set.

In the following context, in order to facilitate further formulation, we will let r_i be "∞" for denoting $0 \le p_i \le u_i$.

EXAMPLE 3.1. The iteration space of a loop nest L_2 is shown in Fig. 1. Assume that R_1 and R_2 are boundary block computation sets. They are denoted as $[\infty, 3]$ and $[2, -3]$, respectively. And, the computation sets that correspond to the regions S_1 and S_2 must not be boundary block computation sets.

DEFINITION 4. *Maximal boundary block computation sets*: Let $\mathrm{expand}(b_i) = b_i + 1$ if $b_i > 0$, and $\mathrm{expand}(b_i) = b_i - 1$ if $b_i < 0$. A boundary block computation set $B = [b_1, b_2, ..., b_n]$ is *maximal* iff $\forall i, 1 \le i \le n$, $[b_1, ..., \mathrm{expand}(b_i), ..., b_n]$, which is a proper superset of B, is not an IICS (i.e., it will include a computation that is not initially independent).

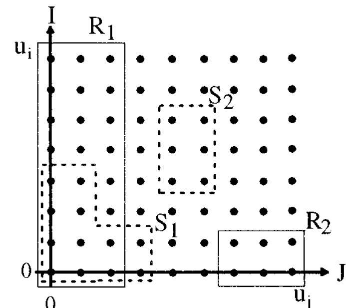

FIG. 1. Example of boundary block computation sets.

In other words, Definition 4 says that B is the block-shaped IICS that has been "expanded" as much as possible.

B. Formulation of the Basic Steps

EXAMPLE 3.2. Consider the loop nest $L_2(V, D)$ with $D = \left[\begin{smallmatrix} 3 & 2 \\ 2 & -3 \end{smallmatrix}\right]$ and $V = \{(i, j)^T | 0 \leq i \leq u_i, 0 \leq j \leq u_j\}$. From Fig. 2(a), it is easy to see that the set $R_{d_1} = [3, \infty] \cup [\infty, 2]$ is an initially independent computation set induced by the dependence vector $\mathbf{d}_1 = (3, 2)^T$, because any two computations in R_{d_1} are independent of each other with respect to $\mathbf{d}_1$. Similarly, $R_{d_2} = [2, \infty] \cup [\infty, -3]$ are the initially independent computation sets induced by the dependence vector $\mathbf{d}_2 = (2, -3)^T$. Since the initially independent computation set R_0, must comply with both the dependence vectors $\mathbf{d}_1$ and $\mathbf{d}_2$, R_0 is equal to the intersection of R_{d_1} and R_{d_2}. That is (as shown in Fig. 2(b)),

$$R_0 = R_{d_1} \cap R_{d_2} = ([3, \infty] \cup [\infty, 2]) \cap ([2, \infty] \cup [\infty, -3])$$
$$= [2, \infty] \cup [3, -3] \cup [2, 2] = [2, \infty] \cup [3, -3].$$

Note that there are two maximal boundary block computation sets, $[2, \infty]$ and $[3, -3]$. (This will be proved later in Theorem 3.2.)

LEMMA 3.1. *For the loop nest $L_n(V, D)$ which has a dependence vector $\mathbf{d}_i = (d_{1i}, d_{2i}, ..., d_{ni})^T$,*

$$R_{d_i} = [d_{1i}, \infty, \infty, ...] \cup [\infty, d_{2i}, \infty, ...] \cup \cdots \cup [\infty, \infty, ..., d_{ni}]$$

is an initially independent computation set with respect to $\mathbf{d}_i$. ($[d_{1i}, \infty, \infty, ...]$, ..., $[\infty, \infty, ..., d_{ni}]$ are boundary block computation sets defined in Definition 3.)

Note that if $d_{ji} \geq u_j$, $1 \leq j \leq n$ (u_i is the upper bound of the ith loop), then we can let $d_{ji} = u_j$.

LEMMA 3.2. *A computation $\mathbf{v}$ of a loop nest L_n is an initially independent computation with respect to the dependence vector $\mathbf{d}_i$ iff $\mathbf{v}$ is in R_{d_i}.*

For brevity's sake, we omit the proofs of the above two lemmas.

THEOREM 3.1. *For a loop nest L_n with m dependence vectors $\mathbf{d}_i$, $1 \leq i \leq m$,*

$$R_0 = \bigcap_{i=1}^{m} R_{d_i}$$

is the maximal initially independent computation set of L_n.

Proof. Complying with every dependence vector $\mathbf{d}_i$, $1 \leq i \leq m$, R_0, by Lemma 3.1, is an initially independent computation set of L_n with respect to D (i.e., all $\mathbf{d}_i$'s). Let $\mathbf{v}$ be an initially independent computation with respect to D. Assume $\mathbf{v} \notin R_0$. Then, $\exists\ \mathbf{d}_i$, $1 \leq i \leq m$, such that $\mathbf{v} \notin R_{d_i}$. This is a contradiction to Lemma 3.2. Hence, any initially independent computation with respect to D must belong to R_0. That is, R_0 aggregates *all* the initially independent computations of L_n and is maximal. ■

The following procedure can derive the maximal initially independent computation set.

PROCEDURE IICS.
Input: L_n with dependence matrix $D = [\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_m]$
($\mathbf{d}_i = (d_{1i}, d_{2i}, ..., d_{ni})^T$, $1 \leq i \leq m$)
Output: R_0
Begin
$R_{d_1} = [d_{11}, \infty, \infty, ...] \cup [\infty, d_{21}, \infty, ...] \cup \cdots \cup [\infty, \infty, ..., d_{n1}]$;
$R_{d_2} = [d_{12}, \infty, \infty, ...] \cup [\infty, d_{22}, \infty, ...] \cup \cdots \cup [\infty, \infty, ..., d_{n2}]$;
$\cdots$
$R_{d_m} = [d_{1m}, \infty, \infty, ...] \cup [\infty, d_{2m}, \infty, ...] \cup \cdots \cup [\infty, \infty, ..., d_{nm}]$;
$R_0 = R_{d_1} \cap R_{d_2} \cap \cdots \cap R_{d_m}$;
Return(R_0);
End.

According to Definition 3, the intersection and combination of two boundary block computation sets ($X = [x_1, x_2, ..., x_n]$ and $Y = [y_1, y_2, ..., y_n]$) are formulated as follows. (Note that $\infty > 0$, since it stands for the loop upper bound.)

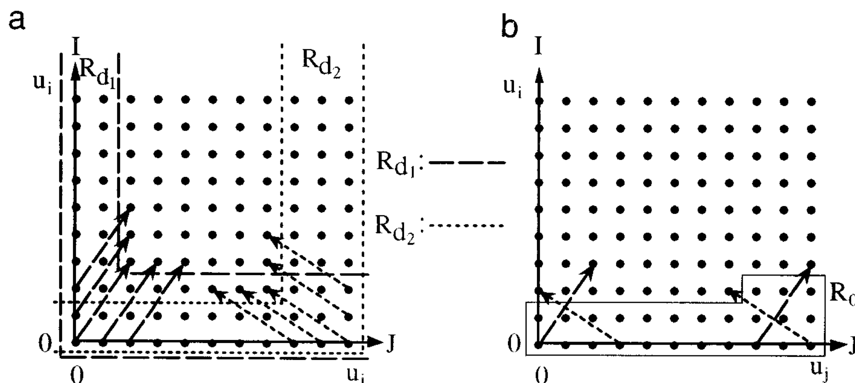

FIG. 2. Initially independent computation sets of Example 3.2.

(a) $S = X \cap Y = [s_1, s_2, ..., s_n]$: $\forall i$, $1 \le i \le n$,

(1) if $x_i \cdot y_i \le 0$ then $s_i = 0$; (Thus, S is an empty set.)

(2) if $x_i > 0$ and $y_i > 0$ then $s_i = \min(x_i, y_i)$; and

(3) if $x_i < 0$ and $y_i < 0$ then $s_i = \max(x_i, y_i)$.

(b) $S = X \cup Y = [s_1, s_2, ..., s_n]$: ($x_i$ "subsumes" y_i if $x_i = \infty$ or if $x_i \cdot y_i > 0$ and $\text{abs}(x_i) > \text{abs}(y_i)$.)

(1) if $\forall i$, $1 \le i \le n$, x_i subsumes y_i then $S = X$;

(2) if $\forall i$, $1 \le i \le n$, y_i subsumes x_i then $S = Y$; and

(3) in the other conditions, X and Y cannot be merged.

COROLLARY 3.1. *The output of Procedure IICS, R_0, must not be empty. Let $R_0 = \bigcup_{h=1}^{k} B_h$, $k \in Z^+$ (k is a constant), then $\forall i$ and j, $1 \le i, j \le k$, $i \ne j$, $B_i \not\subset B_j$.*

COROLLARY 3.2. *In Corollary 3.1, let $B_h = [b_{1h}, b_{2h}, ..., b_{nh}]$, then $\forall h$ and j, $1 \le h \le k$, $1 \le j \le n$, $b_{jh} = \infty$ or $0 < \text{abs}(b_{jh}) \le \max_{i=1}^{m} (\text{abs}(d_{ji}))$. (abs() is the absolute value function.)*

Again, for brevity's sake, we omit the proofs of the above two corollaries.

THEOREM 3.2. *The output of Procedure IICS, R_0, is the union of all the maximal boundary block computation sets.*

Proof. Let $R_0 = \bigcup_{h=1}^{k} B_h$, $k \in Z^+$. $\forall h$, $1 \le h \le k$, if B_h is not a maximal boundary block computation set, then there exists an IICS, say B_x, which is a proper superset of B_h (see Definition 4). Since $\forall i$ and j, $1 \le i, j \le k$, $i \ne j$, $B_i \not\subset B_j$ (Corollary 3.1), B_x is not in R_0. However, B_x is an IICS. This contradicts that R_0 aggregates *all* initially independent computations. So, $\forall h$, $1 \le h \le k$, B_h is a maximal boundary block computation set. Besides, since *any* maximal boundary block computation set includes only initially independent computations, it must belong to R_0. ■

IV. TILING LOOPS FOR EXPLOITING PARALLELISM

DEFINITION 5. *Valid independent computation sets*: For a loop nest L, an independent computation set (ICS) R is *valid* iff L can be partitioned into blocks so that each of them is disjunctive, atomic and identical by translation to R [8]. (R is said to be *invalid* if it is not valid.) We call such a partitioning "*tiling L with R*."

Being disjunctive, each computation is executed exactly only once; and, being atomic, no two blocks are cyclically dependent on each other [8].

DEFINITION 6. *Block dependence vectors*: After the loops are tiled, since all blocks are identical by translation, each of them can be represented by an arbitrary point within it (the so-called tile origin) [1]. Thus, the tile origins define a lattice. We call the dependence vectors in this lattice *block dependence vectors*. Let the loops be tiled with $B = [b_1, b_2, ..., b_n]$. For an original dependence vector $\mathbf{d}_i = (d_{1i}, d_{2i}, ..., d_{ni})^T$, the corresponding block dependence vectors $\mathbf{e}_i = (e_{1i}, e_{2i}, ..., e_{ni})^T$ can be derived as follows [19]. $\forall j$, $1 \le j \le n$,

$$\text{if } b_j = \infty, \text{ then } e_{ji} = 0; \text{ otherwise } (b_j \ne \infty), e_{ji} = \lfloor d_{ji}/\text{abs}(b_j) \rfloor \text{ or } e_{ji} = \lceil d_{ji}/\text{abs}(b_j) \rceil, \tag{4.1}$$

where $d_{ji}/\text{abs}(b_j)$ denotes dividing d_{ji} by the absolute value of b_j. (Note that for d_{ji} there are two values for the corresponding e_{ji}.)

Theorem 3.2 shows that R_0 is the union of all the maximal boundary block computation sets. So, among them, the maximal boundary block computation set that is valid and has the largest size can be chosen to tile the original loops for maximizing parallelism. This is illustrated in the following example.

A. An Illustrative Example

EXAMPLE 4.1. Consider the nested loops $L_3(V, D)$ with $V = \{(i, j, k)^T | 0 \le i, j, k \le \infty\}$ and

$$D = \begin{bmatrix} 1 & 2 & 0 \\ -2 & 4 & 4 \\ 4 & -1 & 3 \end{bmatrix}.$$

With Procedure IICS, $R_0 = ([1, \infty, \infty] \cup [\infty, -2, \infty] \cup [\infty, \infty, 4]) \cap ([2, \infty, \infty] \cup [\infty, 4, \infty] \cup [\infty, \infty, -1]) \cap ([\infty, 4, \infty] \cup [\infty, \infty, 3]) = [1, 4, \infty] \cup [\infty, 4, 4] \cup [2, \infty, 3]$. We intend to tile the loops with a valid independent computation set with maximal size (say, B_t). Since all $[1, 4, \infty]$, $[\infty, 4, 4]$, and $[2, \infty, 3]$ are the maximal boundary block computation sets, B_t can be $[1, 4, \infty]$, $[\infty, 4, 4]$, or $[2, \infty, 3]$. Each case is discussed as follows:

(1) $B_t = [1, 4, \infty]$: Let the loop nest be tiled with $[1, 4, \infty]$. As shown in Fig. 3(a), this is to strip-mine dimensions 1 and 2 with strip lengths 1 and 4, respectively. (Dimension 3 is not strip-mined.) After the loops are tiled, by using Eq. (4.1), in the lattice of tile origins the block dependence vectors are $\mathbf{e}_1^1 = (1, 0, 0)^T$, $\mathbf{e}_1^2 = (1, -1, 0)^T$, $\mathbf{e}_2 = (2, 1, 0)^T$, and $\mathbf{e}_3 = (0, 1, 0)^T$ (see Fig. 3(b)). In the sense that tiling is a transformation of the original loops, we consider $\mathbf{e}_1^1, \mathbf{e}_1^2$, $\mathbf{e}_2$, and $\mathbf{e}_3$ the transformed dependence vectors. Since all the transformed dependence vectors are lexicographically positive [24], such a tiling is valid.

(2) $B_t = [\infty, 4, 4]$: As shown in Fig. 3(c), after the loop are tiled, the block dependence vectors are $\mathbf{e}_1^1 = (0, -1, 1)^T$, $\mathbf{e}_1^2 = (0, 0, 1)^T$, $\mathbf{e}_2^1 = (0, 1, -1)^T$, $\mathbf{e}_2^2 = (0, 1, 0)^T$, and $\mathbf{e}_3 = (0, 1, 1)^T$ (see Fig. 3(d)). Since $\mathbf{e}_1^1 = -\mathbf{e}_2^1$, two blocks are cyclically dependent on each other. This is the so-called dead-locked condition [8]. So, the loops cannot be tiled with $[\infty, 4, 4]$. In fact, such a tiling is invalid because not all $\mathbf{e}$'s are lexicographically positive. (Lexicographically positive dependence vectors will never cause dead-lock.)

We should check if any subset of $[\infty, 4, 4]$ is valid. In the original dependence vectors, there are negative elements in the second dimension. So, properly strip-mining the first

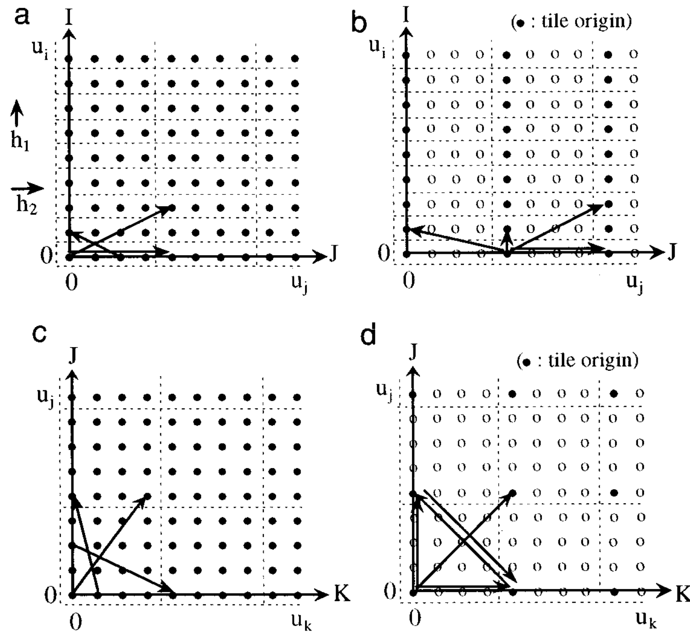

FIG. 3. Illustration of Example 4.1. (The unbounded dimension is not shown.)

dimension will render all **e**'s lexicographically positive. For example, the loops can be correctly tiled with B_t = [1, 4, 4]. (Later, we will formulate how to find the largest subset of B_t when B_t is invalid.)

(3) B_t = [2, ∞, 3]: As the same reasoning in (1), the loop nest can be tiled with [2, ∞, 3].

So we have B_t = [1, 4, ∞], B_t = [1, 4, 4], or B_t = [2, ∞, 3]. Since there are $4 \times u_k$ computations in [1, 4, ∞], 16 computations in [1, 4, 4], and $6 \times u_j$ computations in [2, ∞, 3], we can choose B_t = [2, ∞, 3] to maximize parallelism by assuming that $6 \times u_j$ is greater than 16 and $4 \times u_k$ (u_j and u_k are the loop upper bounds). With B_t = [2, ∞, 3], the original loops can be partitioned into the following parallel form:

```
  Do 100 SI = 0, ⌊UI/2⌋
   Do 100 SK = 0, ⌊UK/3⌋
    DoAll 100 I = SI × 2,min(SI × 2 + 1, UI)
     DoAll 100 J = 1,UJ
      DoAll K = SK × 3, min(SK × 3 + 2,UK)
          loop body
100  Continue
```

B. General Formulation

Given a loop nest L_n, Procedure IICS can derive the maximal initially independent computation set R_0. Let $R_0 = \bigcup_{h=1}^{k} B_h$, $k \in Z^+$, where $B_h = [b_{1h}, b_{2h}, ..., b_{nh}]$ is a maximal boundary block computation set. Whether B_h, $1 \le h \le k$, is a valid ICS is checked as follows. Assume that the loops are tiled with B_h. For each original dependence vector $\mathbf{d}_i = (d_{1i}, d_{2i}, ..., d_{ni})^T$, the corresponding block dependence vector $\mathbf{e}_i$ is computed as Eq. (4.1). If all the $\mathbf{e}_i$'s are lexicographically positive, B_h is a valid ICS. Otherwise, (some $\mathbf{e}_i$ is not lexicographically positive), $\forall j$, $1 \le j \le n$, we may "shrink" b_{jh} (and recalculate every $\mathbf{e}_i$, $1 \le i \le m$) so as to make every $\mathbf{e}_i$ lexicographically positive. There are two cases. (1) $b_{jh} = \infty$: The possible values of b_{jh} that need to be tried are ∞, $\max_{i=1}^{m}(\text{abs}(d_{ji}))$, $\max_{i=1}^{m}(\text{abs}(d_{ji})) - 1$, ..., and 1, since when $b_{jh} > \max_{i=1}^{m}(\text{abs}(d_{ji}))$, $\forall i$, $1 \le i \le m$, $e_{ji} = 0$. (See Eq. (4.1).) (2) $b_{jh} \ne \infty$: The possible values of b_{jh} are b_{jh}, $b_{jh} - 1$, $b_{jh} - 2$, ..., and 1 (If $b_{jh} < 0$, they are b_{jh}, $b_{jh} + 1$, $b_{jh} + 2$, ..., and −1.). Among them, the one with the largest absolute value that makes every $\mathbf{e}_i$ lexicographically positive is chosen. Note that $\forall j$, $1 \le j \le n$, if $b_{jh} = \infty$, we do not have to try $b_{jh} = \max_{i=1}^{m}(\text{abs}(d_{ji})) + 1$, $b_{jh} = \max_{i=1}^{m}(\text{abs}(d_{ji})) + 2$, ..., and so on; and if $b_{jh} \ne \infty$, the possible value of b_{jh} is bounded by $\max_{i=1}^{m}(\text{abs}(d_{ji}))$ (see Corollary 3.2).

Finally, among all the valid ICS's, the one with the largest size (for maximizing parallelism) is chosen to tile the original loops. So we have Procedure MaxValidICS as shown in Fig. 4.

In sum, to tile a loop nest L_n, the first step is to use Procedure IICS to compute R_0, and then the second step is to use Procedure MaxValidICS to find a valid ICS, B_l, whose size is maximal. Thus, L_n can be tiled with B_l. This is described in the following procedure.

PROCEDURE TILE (L_n).
Begin
  Call Procedure IICS; /* derive R_0*/
  Call Procedure MaxValidICS; /* derive B_l */
  Tile the loops L_n with B_l;
End.

C. Applying Transformations to Loops

The loop transformation theory proposed by Wolf and Lam [24] provides a foundation for solving the open prob-

```
Procedure MaxValidICS /* to find the valid ICS whose size is maximal */
  Input: R0 = ∪_{h=1}^{k} Bh (Bh=[b1h, b2h, ···, bnh]), k∈Z+ and dependence vectors di (1≤i≤m)
  Output: Bl
Begin
    For h=1 to k
Step1: {  For i = 1 to m
             For j=1 to n
             { if (bjh=∞) then eji=0;
                 else { e1ji=⌊dji/abs(bjh)⌋, e2ji=⌈dji/abs(bjh)⌉; } }
           If (∀ i, 1≤i≤m, ei is lexicographically positive) then
              mark Bh as valid;
           else
Step2:      { Bh = a proper subset of Bh; /* as described in Section IV.B */
              if (|Bh|>1) goto Step1; }
        }
Step3:  Among all the valid Bh's, choose the one with the largest size (say Bl)
        Return(Bl)
End.
```

FIG. 4. Procedure MaxValidICS.

lem of how to combine reversal, permutation, skewing transformations and tiling techniques for the goal of maximizing parallelism. They show that any uniform dependence loop nest can be transformed (by skewing) into a canonical form—fully permutable loop nest (i.e., the dependence vectors have no negative elements).

THEOREM 4.1. *Consider a fully permutable loop nest L_n with m dependence vectors $\mathbf{d}_i = (d_{1i}, d_{2i}, \ldots, d_{ni})^T$, $1 \le i \le m$. If the wavefront transformation*

$$T_W = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

is applied to L_n, then $B_t = [\min_{i=1}^{n}(d_{1i} + d_{2i} + \cdots + d_{ni}), \infty, \infty, \ldots] \subset R_0$ is the output of Procedure MaxValidICS, where R_0 is the maximal IICS derived from Procedure IICS.

Proof. After the wavefront transformation T_W is applied to L_n, the dependence vectors are transformed into $T_W \cdot \mathbf{d}_i = (d_{1i} + d_{2i} + \cdots + d_{ni}, d_{1i}, d_{2i}, \ldots, d_{(n-1)})^T$, $1 \le i \le m$. In Procedure IICS, clearly, $B_t = [\min_{i=1}^{m}(d_{1i} + d_{2i} + \cdots + d_{ni}), \infty, \infty, \ldots] \subset R_0$. Let $R_0 = \bigcup_{h=1}^{k} B_h$, $k \in Z^+$. If $\exists g$, $1 \le g \le k$, $|B_g| > |B_t|$, then $B_g = [y, \infty, \infty, \ldots]$ where $y \in Z$, $y > \min_{i=1}^{m}(d_{1i} + d_{2i} + \cdots + d_{ni})$. ($|B_g|$ and $|B_t|$ denote the number of computations in B_g and B_t, respectively.) Obviously, $(y, \infty, \infty, \ldots)$ is not an initially independent computation and hence $(y, \infty, \infty, \ldots) \notin R_0$. This implies that $B_g \not\subset R_0$, which is a contradiction to $R_0 = \bigcup_{h=1}^{k} B_h$. So, $\forall h$, $1 \le h \le k$, $|B_t| \ge |B_h|$. In Procedure MaxValidICS, since $\forall i, j$, $1 \le i \le m$, $1 \le j \le n$, $d_{ji} \ge 0$ (L is fully permutable), all $\mathbf{e}_i$'s are lexicographically positive. So, every B_h, $1 \le h \le k$, is a valid ICS. Since $\forall h$, $1 \le h \le k$, $|B_t| \ge |B_h|$ and $B_t \subset R_0 = \bigcup_{h=1}^{k} B_h$, the output of Procedure MaxValidICS is B_t. ∎

Note that, in Theorem 4.1, the size of B_t is always greater than one. And we can apply any other wavefront transformation that is a permutation of T_W to the original loop nest to derive a valid ICS, because, in Procedure MaxValidICS, $d_{ji} \ge 0$ ($1 \le i \le m$, $1 \le j \le n$) implies that $\mathbf{e}_i$ is lexicographically positive whether $\mathbf{e}_i$ is permuted or not.

D. Partitioning Loops with Combined ICS's

A loop nest whose all dependence vector elements are positive can be regularly partitioned with some combined valid ICS's as described as follows.

Consider a loop nest L_n with m dependence vectors $\mathbf{d}_i = (d_{1i}, d_{2i}, \ldots, d_{ni})^T$ where $\forall i, j$, $1 \le i \le m$, $1 \le j \le n$, $d_{ji} > 0$. In Procedure IICS, clearly, $[\min_{i=1}^{m}(d_{1i}), \infty, \ldots, \infty]$ is a subset of $[d_{1i}, \infty, \infty, \ldots]$, $1 \le i \le m$. That is, it is a

subset of R_{d_i}, $1 \le i \le m$. Since $R_0 = \bigcap_{i=1}^{m} R_{d_i}$, $[\min_{i=1}^{m}(d_{1i}), \infty, ..., \infty]$ is a subset of R_0 and therefore is an IICS. By the same reasoning, all $[\infty, \min_{i=1}^{m}(d_{2i}), \infty, ..., \infty]$, ..., and $[\infty, ..., \min_{i=1}^{m}(d_{ni})]$ are IICS's. And it is easy to see that all $[\min_{i=1}^{m}(d_{1i}), \infty, ..., \infty]$, ..., $[\infty, ..., \min_{i=1}^{m}(d_{ni})]$ are valid. The following example illustrates how to combine them to tile the loops.

EXAMPLE 4.2. Consider the loop nest $L_2(V, D)$ with $D = \left[\begin{smallmatrix} 2 & 3 \\ 3 & 2 \end{smallmatrix}\right]$. Let $B_t = [\min(2,3), \infty] \cup [\infty, \min(3,2)] = [2, \infty] \cup [\infty, 2]$. Then the loops can be partitioned as shown in Fig. 5.

Clearly, this can be easily generalized to multidimensional loops. For brevity's sake we omit the general formulation.

V. COMPARISONS WITH RELATED WORK

A. Comparisons with Multiple Hyperplane Partitioning

Irigoin and Triolet [8] propose a general formulation called supernode partitioning which formulates a condition for finding n families of parallel hyperplanes for tiling n-fold nested loops. The vectors normal to the n families of hyperplanes, say $\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_n$, should be linearly independent. Let H be the matrix whose rows are the vectors $\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_n$. Thus, the tiling condition is $HD \ge \mathbf{0}$ (D is the dependence matrix). With this condition, admissible tiling can be determined [8]. However, it does not provide quantitative procedures for choosing the size and shape of tiles. Based on the condition $HD \ge \mathbf{0}$, some automatic blocking methods for determining suitable sizes and shapes of tiles are further studied by other research, including the works done by Schreiber and Dongarra [20], and Ramanujam and Sadayappan [18]. Since all these methods use multiple hyperplanes to partition loops, we refer to them as the multiple hyperplane partitioning methods.

Some points about the comparison between the multiple hyperplane partitioning methods and the proposed method are discussed as follows. First, some valid tiling does not meet the condition $HD \ge \mathbf{0}$. Consider Example 4.1 again. From Fig. 3(a), we can see that the vectors normal to the

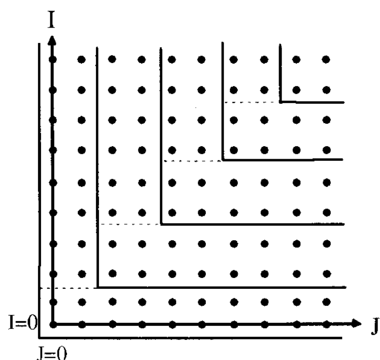

FIG. 5. Illustration of Example 4.2.

partitioning hyperplanes are $\mathbf{h}_1 = (x, 0, 0)$, $\mathbf{h}_2 = (0, y, 0)$, $\mathbf{h}_3 = (0, 0, 0)$ $(x, y \in Z^+)$ and hence

$$H = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

So, the condition $HD \ge \mathbf{0}$ does not hold. However, as has been shown in Example 4.1, $[1, 4, \infty]$ can correctly tile the loops. This is because H is rank-deficient and leads to an infinite tile [7]. (If H is a full rank matrix, then it can be used as an exact condition.) Therefore, to correctly tile a loop nest, the condition $HD \ge \mathbf{0}$ is too strong. In contrast, in the proposed method, the tiling condition is that a tiling is valid if and only if *all* the resulting block dependence vectors are lexicographically positive. Clearly, this condition is exact since it conforms to the basic concept of valid dependence vectors [24].

Second, the multiple hyperplane partitioning methods focus on building admissible tiles but do not address on how to maximize parallelism within tiles. Partitioning loops with them, the computations in a tile are not necessarily independent. In contrast, in this paper, the proposed method aims at aggregating independent computations into a tile; and it provides a systematic procedure to maximize the tile size (and hence maximize parallelism).

Third, unlike the multiple hyperplane partitioning methods that partition loops into polyhedra, the proposed method partitions loops into rectangular blocks. Such a partitioning make the parallel code generation very easy. And, most important, all the available parallelism (initially independent computations) can be aggregated into rectangular blocks (Theorem 3.2). From this point of view, partitioning loops into rectangular blocks is useful enough for maximizing parallelism. In contrast, in the multiple hyperplane partitioning methods, code generation is not straightforward and needs some efforts [1]. Also, as shown in [17], this is related to the issue of code generation for nonunimodular transformations [12].

B. Comparisons with Cycle Shrinking

The cycle shrinking method [16] can be seen as a special case of affine scheduling [3]. Like the proposed method of this paper, the cycle shrinking method also partitions nested loops into blocks (except for true dependence shrinking) that contain only independent computations.

EXAMPLE 5.1. Consider the following loop nest [16]:

```
Do 100 I = 3,N1
 Do 100 J = 5,N2
   A(I,J) = B(I-3,J-3)
   B(I,J) = A(I-2,J-5)
100  Continue
```

Its dependence matrix is $\left[\begin{smallmatrix} 2 & 3 \\ 5 & 3 \end{smallmatrix}\right]$. If simple or selective shrinking is applied, the corresponding ISDG's are shown

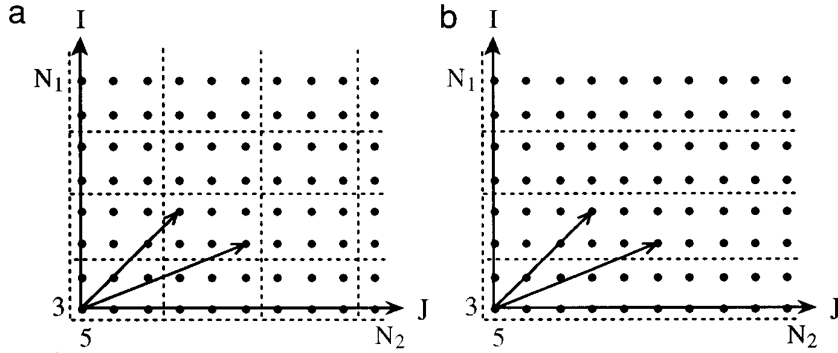

FIG. 6. The ISDG's of simple and selective shrinking for Example 5.1.

in Figs. 6(a) and 6(b), respectively. Clearly, the degrees of parallelism are 6 and $2N_2 - 8$, respectively. If true dependence shrinking is applied, the iteration space is partitioned in an "irregular" fashion [16] and the degree of parallelism is $\min(2(N_2 - 4) + 5, 3(N_2 - 4) + 3)$, which is only a little better than $2N_2 - 8$ of selective shrinking. Obviously, by true dependence shrinking, the loops are not tiled. So we will not further discuss it since, in this paper, we focus on loop tiling.

In contrast, with our proposed method, $R_0 = [3, 5] \cup [2, \infty] \cup [\infty, 3]$. Thus, the original loops can be partitioned with [3, 5] (see Fig. 7(a)) or with $[2, \infty] \cup [\infty, 3]$ (see Fig. 7(b)).

From Figs. 6 and 7, it is clear that more parallelism is exploited with the proposed method than with cycle shrinking.

C. Comparisons with D'Hollander's Method

Some loop partitioning approaches determine the dependences among computations by using the linear combination of dependence vectors. A typical example is the method proposed by D'Hollander [4] (which is an improvement of the minimum distance method [14]). It transforms the dependence matrix D into a triangular matrix D^t, and then, based on D^t, partitions the loops into totally independent groups of computations, which can be executed independently.

The comparisons between our proposed method and D'Hollander's method are as follows. For D'Hollander's method: First, in the transformed codes all the DoAll loops are made outermost. That is, the method can exploit coarse-grain parallelism [24] and the transformed codes are suitable for execution on distributed memory systems since each group can be executed independently. Second, it is shown that the degree of parallelism is limited to the determinant of the dependence matrix [4] and therefore cannot be large. Third, clearly, the loops are not partitioned into regions with regular shapes. This will deteriorate performance in practical implementation.

In contrast, for our proposed method: First, the independent computations are aggregated into a sequence of DoAll loops, which can be executed in parallel. That is, it makes all the DoAll loops innermost and obtains fine-grain parallelism [24]. Therefore, the transformed codes are suitable or execution on shared memory systems, in which the interprocessor communication cost is small. Note that, for a distributed memory system that provides fast interprocessor synchronization and communication mechanism, our method can also apply to such a system for efficient execution. (The iterations of a DoALL loop can be executed on different processors when suitable barrier synchronization is forced among processors.) Second, as shown in this paper, the sizes of the partitioned blocks usually depend on the loop bounds but not limited to a constant. So the degree of parallelism is usually very large. (As shown in Theorem 4.1, if the wavefront transformation is used, then the degree of parallelism must be as large as

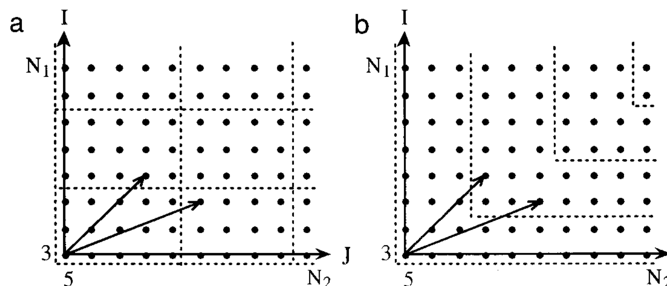

FIG. 7. The ISDG's of the proposed method for Example 5.1.

loop bounds.) Third, the loops are partitioned into regions with regular shape. So the compiler can easily generate the parallel codes and the parallel codes can be efficiently executed.

VI. CONCLUSIONS

In this paper, we study the problem of tiling nested loops with uniform dependences for maximizing parallelism, and propose a new approach to it. At first, based on identifying the initially independent computations, all the available parallelism is exposed. Then, how to tile the loops based on the initially independent computation sets is discussed. We show that all the available parallelism (initially independent computations) can be aggregated into rectangular blocks. So the loops are tiled into rectangular blocks to maximize parallelism. We also show that if the wavefront transformation is combined with the proposed method, the loops can always be tiled so that the tile size is greater than one.

In comparison with some well-known methods, the proposed method is shown to have several advantages: First, since loops are partitioned into equal-sized blocks, the data reference locality and processor load balancing are better exploited. Second, all the procedures are systematic and can be easily implemented in a parallelizing compiler. Third, unlike the multiple hyperplane partitioning methods, which may not be able to find some valid tiling, the procedures described in this paper do not "lose" any possible valid tiling (for rectangular tiles) since they check the basic property of lexicographic positiveness of dependence vectors.

ACKNOWLEDGMENTS

We thank the referees for their valuable comments and suggestions, and Professor T. M. Jiang at the University of Alaska for many enlightening discussions. This research was supported in part by National Science Council, Taiwan, under contract NSC84-2213-E-002-001.

REFERENCES

1. Ancourt, C., and Irigoin, F. Scanning polyhedra with Do loops. *Proc. ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*. 1991, pp. 39–50.
2. Banerjee, U. *Dependence Analysis for Supercomputing*. Kluwer Academic, Boston, 1988.
3. Darte, A., and Robert, Y. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distrib. Systems* **5,** 8 (Aug. 1994), 814–822.
4. D'Hollander, E. H. Partitioning and labeling of loops by unimodular transformations. *IEEE Trans. Parallel Distrib. Systems* **3,** 4 (July 1992), 465–476.
5. Fortes, J. A. B., and Parisi-Presicce, F., Optimal linear schedules for the parallel execution of algorithms. *Proc. Int. Conf. on Parallel Processing*. 1984, pp. 322–329.
6. Gannon, D., Jalby, W., and Gallivan, K. Strategies for cache and local memory management by global program transformation. *J. Parallel Distrib. Comput.* **5,** 10 (Oct. 1988), 587–616.
7. Irigoin, F. Partitionnement des boucles imbriquées: Une technique d'optimisation pour les programmes scientifiques. Ph.D. thesis, Université de Paris VI, Paris, 1987.
8. Irigoin, F., and Triolet, R. Supernode partitioning. *Proc. 15th Annual ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages*. 1988, pp. 319–329.
9. King, C., and Ni, L. M. Grouping in nested loops for parallel execution on multicomputers. *Proc. Int. Conf. Parallel Processing*, Vol. II. 1989, pp. 31–38.
10. Kuck, D. J., Muraoka, Y., and Chen, S.-C. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Trans. Comput.* **21**, 12 (Dec. 1972), 1293–1310.
11. Lamport, L. The parallel execution of do loops. *Comm. ACM* **17,** 2 (Feb. 1974), 83–93.
12. Li, W., and Pingali, K. A singular loop transformation framework based on non-singular matrices. *J. Parallel Programming* **22,** 2 (Apr. 1994), 183–205.
13. Nicolau, A. Loop quantization: A generalized loop unwinding technique. *J. Parallel Distrib. Comput.* **5,** 10 (Oct. 1988), 568–586.
14. Peir, J.-K., and Cytron, R. Minimum distance: A method for partitioning recurrences for multiprocessors. *IEEE Trans. Comput.* **38,** 8 (Aug. 1989), 1203–1211.
15. Polychronopoulos, C. D. *Parallel Programming and Compilers*. Kluwer Academic, Dordrecht/Norwell, MA, 1988.
16. Polychronopoulos, C. D. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Trans. Comput.* **27,** 8 (Aug. 1988), 991–1004.
17. Ramanujam, J. Non-unimodular transformations of nested loops. *Proc. Supercomputing '92*. 1992, pp. 214–223.
18. Ramanujam, J., and Sadayappan, P. Tiling multidimensional iteration space for multicomputers, *J. Parallel Distrib. Comput.* **16,** 2 (Oct. 1992), 108–120.
19. Risset, T. A method to synthesize modular systolic arrays with local broadcast facility. *Proc. Int. Conf. Application Specific Array Processors*. 1992, pp. 415–428.
20. Schreiber, R., and Dongarra, J. Automatic blocking of nested loops. Tech. Rep. CS-90–108, Univ. of Tennessee, Knoxville, TN, 1990.
21. Shang, W., and Fortes, J. A. B. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Trans. Comput.* **40,** 6 (Jun. 1991), 723–742.
22. Shang, W., and Fortes, J. A. B. Independent partitioning of algorithms with uniform dependencies. *IEEE Trans. Comput*. **41,** 2 (Feb. 1992), 190–206.
23. Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. *Proc. ACM SIGPLAN 1991 Conf. Programming Language Design Implementation*. 1991, pp. 30–44.
24. Wolf, M. E., and Lam, M. S. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Systems* **2,** 4 (Oct. 1991), 452–471.
25. Wolfe, M. More iteration space tiling. *Proc. Supercomputing'89*. 1989, pp. 655–664.
26. Wolfe, M. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, 1989.
27. Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers*, Addison–Wesley, New York, 1990.

YEONG-SHENG CHEN received the B.E. and M.E. in computer engineering from National Chiao-Tung University, Hsinchu, Taiwan, in 1988 and 1990, and the Ph.D. degree in electrical engineering from National Taiwan University, Taipei, Taiwan in 1996. Currently, he is an associate professor at the Department of Electronics Engineering, Hwa-Hsia College of Technology, Taipei, Taiwan. His research interests in-

clude compiler techniques to improve parallelism, program restructuring, and scheduling.

SHENG-DE WANG received the B.E. from National Tsing Hua University, Hsinchu, Taiwan, in 1980, and the M.E. and the Ph.D. in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1982 and 1986, respectively. In 1986 he joined the faculty of the Department of Electrical Engineering at National Taiwan University, Taipei, Taiwan, where he is currently a professor. His research interests include parallel processing, artificial intelligence, and neurocomputing. Dr. Wang is a member of the Association for Computing Machinery, the International Neural Networks Society, and the IEEE Computer Society. He is also a member of the Phi Tau Phi Honor society.

CHIEN-MIN WANG received the B.E. and the Ph.D. in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1987 and 1991, respectively. Then he joined the Institute of Information Science, Academia Sinica, Taipei, Taiwan, as an assistant research fellow. Currently, he is an associate research fellow at the same institute. His major research interest includes parallelizing compilers, parallel algorithms, parallel computer architectures, and object-oriented programming. He is a member of the IEEE Computer Society.

Received January 10, 1994; revised June 12, 1995; accepted November 16, 1995