Parallelism for Nested Loops with Simple Subscripts

Sam Jin Jeong*
Division of Information and Communication Engineering, BaekSeok University
Anseo-dong 115, Cheonan City, Korea 330-704

ABSTRACT

In this paper, we propose improved loop splitting method for maximizing parallelism of single loops with non-constant dependence distances. By using the iteration and distance for the source of the first dependence, and by our defined theorems, we present generalized and optimal algorithms for single loops with non-uniform dependences (MPSL). By the extension of the MPSL method, we also apply to exploit parallelism from nested loops with simple subscripts, based on cycle shrinking and loop interchanging method. The algorithms generalize how to transform general single loops with non-uniform dependences as well as nested loops with simple subscripts into parallel loops.

Keywords: *Parallelizing Compiler, Loop Splitting, Nested Loop, Multiple Dependences, Non-uniform Dependences*

1. INTRODUCTION

Partitioning of loops requires efficient and exact data dependence analysis [1][2]. A precise dependence analysis helps in identifying dependent/independent iterations of a loop. And it is important that appropriate dependence analysis be applied to exploit maximum parallelism within loops. We can consider some tests that examine the dependence of one-dimensional subscripted variables – the separability test, the GCD test and the Banerjee test [5][7]. In general, the GCD test is applied first because of its simplicity, even if it is an approximate test. Next, for the case that the gcd test is true, the separability test is attempted again, and through this exact test, it can be obtained additional information such as solution set, and minimum and maximum distances of dependence, as well as whether the existence of dependence or not.

When we consider the approach for single loops, we can review two partitioning techniques proposed in [3] which are fixed partitioning with minimum distance and variable partitioning with ceil(d*(i)*). However, these leave some parallelism unexploited, and the second case has some constraints.

The rest of this paper is organized as follows. Chapter two describes our loop model, and introduces the concept of data-dependence computation in actual programs. In chapter three, we review some partitioning techniques of single loops such as loop splitting method by thresholds and Polychronopoulos' loop splitting method. In chapter four, we propose a generalized and optimal method to make the iteration space of a loop into partitions with variable sizes (MPSL). We also apply to exploit parallelism from nested loops with simple subscripts by the extension of the MPSL method. The algorithms generalize how to transform general single loops with non-uniform dependences as well as nested loops with simple subscripts into parallel loops. Finally, we conclude in chapter five with the direction to enhance this work.

2. PROGRAM MODEL AND DATA DEPENDENCE ANALYSIS

For data-dependence computation in actual programs, the most common situation occurs when we are comparing two variables in a single loop and those variables are elements of a one-dimensional array, with subscripts linear in the loop index variable. Then this kind of loop has a general form is shown in figure 1. Here, l, u, a_1, a_2, b_1 and b_2, are integer constants known at compile time.

DO $I = l, u$
S_1: $A(a_1*I + a_2) = \cdots$
S_2: $\cdots = A(b_1*I + b_2)$
END

Fig. 1 A single loop model.

For dependence between statements S_1 and S_2 to exist, we must have an integer solution (i, j) to equation (1) that is a linear diophantine equation in two variables. The method for solving such equations is well known and is based on the extended Euclid's algorithm [4].

$$a_1 i + a_2 = b_1 j + b_2 \text{ where } l \le i, j \le u \quad (1)$$

This equation may have infinitely many solutions (i, j)

* *Corresponding author. E-mail : sjjeong@bu.ac.kr*
Manuscript received Sep. 30, 2008 ; accepted Oct. 21, 2008

given by a formula of the form:

$$(i, j) = ((b_1/g)t + i_1, (a_1/g)t + j_1$$
$$\text{where } (i_1, j_1) = ((b_2\text{-}a_2)\, i_0/g, (b_2\text{-}a_2)\, j_0/g) \quad (2)$$

i_0, j_0 are any two integers such that $a_1\, i_0 - b_1 j_0 = g(\gcd(a_1, b_1))$ and t is an arbitrary integer [5][6]. Acceptable solutions are those for which $l \le i, j \le u$, and in this case, the range for t is given by

$$\max(\min(\alpha, \beta), \min(\gamma, \delta)) \le t \le \min(\max(\alpha, \beta), \max(\gamma, \delta))$$
$$\text{where } \alpha = -(l - i_1)/(b_1/g), \beta = -(u - i_1)/(b_1/g),$$
$$\gamma = -(l - j_1)/(a_1/g), \delta = -(u - i_1)/(a_1/g). \quad (3)$$

3. RELATED WORKS

Now, we review some partitioning techniques of single loops. We can exploit any parallelism available in such a single loop in figure 1, by classifying the four possible cases for a_1 and b_1, coefficients of the index variable I, as given by (4).

(a) $a_1 = b_1 = 0$
(b) $a_1 = 0, b_1 \neq 0$ or $a_1 \neq 0, b_1 = 0$
(c) $a_1 = b_1 \neq 0$
(d) $a_1 \neq 0, b_1 \neq 0, a_1 \neq b_1$ (4)

In case 4(a), because there is no cross-iteration dependence, the resulting loop can be directly parallelized. In the following subsections, we briefly review several loop splitting methods for the cases of 4(b) through 4(d).

3.1 Loop splitting by thresholds

A threshold indicates the number of times the loop may be executed without creating the dependence. In case 4(b), for a dependence to exist, there must be an integer value i of index variable I such that $b_1 * i + b_2 = a_2$ (if $a_1 = 0$) or $a_1 * i + a_2 = b_2$ (if $b_1 = 0$) and $l \le i \le u$. If there is no solution, then there is no cross-iteration dependence and the loop can also be parallelized. And if integer exists, then there exist a flow dependence (or anti-dependence) in the range of I, $[l, u]$ and an anti-dependence (or flow dependence) in $[i, u]$. In this case, be breaking the loop at the iteration $I = i$ (called *turning threshold*), the two partial loops can be transformed into parallel loops.

In case 4(c), let (i, j) be an integer solution to (1), then there exists a dependence in the range of I and the dependence distance (d) is $|j - i| = |(a_2 - b_2)/a_1|$. Here, the loop can be transformed into two perfectly nested loops; a serial outer loop with stride d (called constant threshold) and a parallel inner loop [5].

In case 4(d), an existing dependence is non-uniform since there is a non-constant distance, that is, such that it varies between different instances of the dependence. And we can consider exploiting any parallelism for two cases when $a_1 * b_1 < 0$ and $a_1 * b_1 > 0$. Suppose now that $a_1 * b_1 < 0$. If (i, i) is a solution to (1), then there may be all dependence sources in (l, i) and all dependence sinks in $[i, u]$. Therefore, by splitting the loop at the iteration $I = i$ (called *crossing threshold*), the two partial loops can be directly parallelized [5]. Figure 2(c) shows the general form of loop splitting by the crossing threshold.

```
                                    DOALL I = 1, 4
                                      A(I+5) = ···
DO I = 1, 10                            ··· = A(9)
  A(I+5) = ···            =>        ENDDO
    ··· = A(9)                      DOALL I = 5, 10
ENDDO                                 ···
                                    ENDDO
```
(a) Using turning threshold.

```
                                  DO I' = 1, N, abs((b2-a2)/a1)
DO I = 1, N                         DOALL  I  = I', I'+(abs((b2-a)/a1)-1
  A(a1*I+ a2) = ···                   A(a1*I+ a2) = ···
    ··· = A(b1*I+ b2)     =>              ··· = A(b1*I+ b2)
ENDDO                               ENDDO
                                  ENDDO
```
(b) Using constant threshold.

```
                                  DOALL I = 1, floor((b2 - a2)/ (a1+ b1))+1,N
                                    A(a1*I+ a2) = ···
DO I = 1, N                               ··· = A(b2+ b1*I)
  A(a1*I+ a2) = ···       =>      ENDDO
    ··· = A(b2+ b1*I)             DOALL I = floor((b2 - a2)/ (a1+ b1))+1,N
ENDDO                               ···
                                  ENDDO
```
(c) Using crossing threshold.

Fig. 2 The loop splitting by thresholds.

3.2 Polychronopoulos' loop splitting

We can also consider exploiting any parallelism for the case 4(d) when $a_1 * b_1 \ge 0$. We will consider three cases whether it exists only flow dependence, anti-dependence, or both in the range of I. First, let (i, j) be an integer solution to (1). If the distance, $d(i)$ depending on i, as given by (5), has a positive value, then there exists a flow dependence, and if $d_a(j)$ depending on j, as given by (6), has a positive value, then there exists an anti-dependence. Next, if (x, x) is a solution to (1) (x may not be an integer.), then $d(x) = d_a(x) = 0$ and there may exist a flow (or anti-) and an anti-dependence (or flow) before and after I = ceil(x), and if x is an integer, then there exists a loop-independent dependence at $I = x$. Here, suppose that Then for each value of I, the element $A(a_1 * I + a_2)$ defined by that iteration cannot be consumed before ceil($d(i)$) iterations later, and this indicates that ceil($d(i)$) iterations can execute in parallel.

$$d(i) = j - i = D(i)/b_1, \text{ where } D(i) = (a_2 - b_1) * i + (a_2 - b_1) \quad (5)$$
$$d_a(j) = i - j = D_a(j)/a_1, \text{ where } D_a(j) = (b_1 - a_1) * j + (b_2 - a_2) \quad (6)$$

Consider the loop, as given in figure 3, in which there exist flow dependences. $d(i) = D(i)/b_1 = (i + 5)/2 > 0$ for each value of I and $d(i)$ have integer values, 3, 4, 5, ... as the value of I is incremented.

DO I =1, N

```
S1 :  A(3I+1) = ...
S2 :  ... = A(2I-4)
ENDDO
```

Fig. 3 An example of a single loop.

Figure 4 shows the results of applying two transformations using minimum distance and ceil(d(i)) proposed in [3] to the loop in Fig. 3, respectively. However these leave some parallelism unexploited. Moreover, the transformed loop in figure 4(b) has some constraints: It must be only a flow dependence in the original loop, the first iteration of the original loop must be the iteration at which a dependence source exists, and d(i) ≥ 1 for each value of I.

```
DO I' = 1, N, 3                    I' = 1
  DOALL I = I', min(N, I'+2)       L: inc = min(N-I', ceil((I'+5)/2)-1)
    A(3I+1) = ...                  DOALL I = I'+inc
          ... = A(2I-4)                A(3I+1)= ...
  ENDDO                                  ... =A(2I-4)
ENDDO                              ENDDO
                                   I' = I'+inc+1
                                   If I' < N then goto L
(a) Using minimum distance.        (b) Using ceil(d(i)).
```

Fig. 4 Polychronopoulos' loop splitting.

4. PARALLELISM FOR NESTED LOOPS

From a single loop with non-constant distance such that it satisfies the case (d) in (4) and $a_1*b_1 > 0$, we can get the following Lemmas. For convenience' sake, suppose that there is a flow dependence in the loop.

Lemma 1: The number of iterations between a dependence source and the next source, sd is given by $| b_1 | /g$ iterations where $g = gcd(a_1, b_1)$.

And we can know the facts that if we obtain the iteration for the source of the first dependence, then we can compute the others easily, and $i \equiv j (\mathrm{mod}\ s_d)$ for i, j are arbitrary iterations for all sources.

Lemma 2: The dependence distance, that is, the number of iterations between the source and the sink of a dependence, is $D(i)/b_1$ where $D(i) = (a_1 - b_1) * i + (a_2 - b_2)$, and the increasing rate of a distance per one iteration, d' is given by $(a_1 - b_1)/ b_1$. And the difference between the distance of a dependence and that of the next dependence, d_{inc} is $| a_1 - b_1 | /g$.

Hence, if we obtain the distance for the source of the first dependence, then we can compute the others easily. Also for the case of anti-dependence, similarly, Lemma 1 and 2 can be represented. Namely, s_d is given by $| a_1 | /g$ iterations where $g = gcd(a_1, b_1)$, the distance is given by (6), and d' is $(b_1-a_1)/a_1$. And $d_{inc} = d' * s_d = (b_1-a_1)/a_1 * | a_1 | /g = | b_1 - a_1 | /g$.

By using the iteration and distance for the source of the first dependence, and concepts defined by Lemma 1 and 2, we obtain the generalized and optimal algorithm to maximize parallelism from single loops with non-uniform dependences.

4.1 Transformation of Single Loops

Procedure MaxSplit shows the transformation of single loops satisfying the case (d) in (4) and $a_1*b_1 > 0$ into partial parallel loops [8].

The following Procedure MaxSplit_2 generalizes how to transform general single loops, as shown in figure 1, into parallel loops.

Procedure MaxSplit_2

```
/* A generalized transformation of single loops into parallel loops */
BEGIN
  /* (1) Testing for data dependence. */
Step 1:
  If (gcd test) is not true then
    {Transform a loop into a parallel loop directly; Stop};
  If (separability test) is not true then
    {Transform a loop into a parallel loop directly; Stop};
  /* (2) Data dependence computation and transformation for each of cases
  in (4). */
Step 2:
  If case (a) is true then
    {Transform a loop into a parallel loop directly; Stop};
  If case (b) or case (c) or a1*b1 <0 is true then
      {Compute a turning threshold or a constant threshold or a crossing
      threshold and process the loop splitting by using those, respectively;
      Stop};
Step 3: /* The case satisfying (d) in (4) and a1*b1 >0 */
  If (x, x) is a solution to (1) then
    go to Step 5;
Step 4: /* The case that there exists only a flow dependence (or anti-
    dependence) */
    /* There exists flow dependence */
  If d(i) > 0 for I = i within the range of loop then
    {sd = | b1 | /g; dinc = | a1- b1 | /g};
    /* There exists anti-dependence */
  else {sd = | a1 | /g; dinc = | b1- a1 | /g};
    /* α, β: the iteration and distance for the source of the first dependence
    in the loop, respectively */
    Compute α, β by the separability test;
Step 4.1: Call MaxSplit(I, u, sd , dinc, α, β);
  Stop;
Step 5: /* The case that there exist both flow and anti-dependences in the
    range of loop */
    The same as Step 4 for the range of l ≤ I ≤ ceil(x);
Step 5.1: Call MaxSplit(I,ceil(x), sd , dinc, α, β);
Step 6: The same as Step 4 for the range of ceil(x)+1 ≤ I ≤ u;
Step 6.1: Call MaxSplit(ceil(x) + 1, u, sd , dinc, α, β);
Step 7: Merge the last block splitted by Step 5 and the first block splitted by
    Step 6 together;
  Stop;
END MaxSplit_2
```

In step 5-6, if there exist a flow (or anti-) and an anti-dependence (or flow) before and after I = ceil(x), as defined in step 3, then dividing the loop two parts, and each of parts can be transformed by Procedure MaxSplit. No dependences are violated by the transformed block in step 7, since there may be different type of dependences before and after I = ceil(x) and a loop-independent dependence may exist at $I = x$, even if it exists.

4.2 Transformation of Nested Loops with Simple Subscripts

In previous sections, we proposed a generalized and optimal method for single loops (MPSL) only. This section discusses the extension of the MPSL method, in order that it can be applied to exploit parallelism from nested loops with simple subscripts. However, it is difficult to apply this method to nested loops with coupled subscripts due to irregular and complex dependence constraints. If we consider nested loops with simple subscripts (i.e., the dimensionality of arrays in the loop is equal to the nested loop depth, and the subscript is the linear function of only a corresponding loop variable) as given in figure 5, we can derive an efficient method for these loops from enhancing the MPSL method, based on cycle shrinking [3][9] and loop interchange [10].

Cycle shrinking method is one of methods of extracting parallelism from nested loops with uniform dependences, and there are three types in it: simple cycle shrinking, selective cycle shrinking, and true dependence cycle shrinking. Cycle shrinking is useful when the minimum of the dependences λ is greater than unity; it transforms a sequential DO loop into two perfectly nested loops: a sequential outer loop and a parallel inner loop. For example, let a single loop have u iterations, i.e., index set J={1,2, ⋯ , u}, than iterations in sets {1,2, ⋯ , λ}, or {λ + 1, ⋯ , 2λ}, ⋯ can be executed in parallel without violating data dependence relations. If the algorithm has a cyclic dependence structure, then the size of the dependence cycle is shrunk by a reduction factor λ. More examples for the intuitive concepts behind cycle shrinking can be found in [3].

```
DO I1 = l1, u1
  DO I2 = l2, u2
    ...
      DO In = ln, un
        A(f1(I1), ⋯ , fn(In)) = ⋯
                              ⋯ = A(g1(I1), ⋯ , gn(In))
      ENDDO
    ...
  ENDDO
ENDDO
```

Fig. 5 A type of nested loop with simple subscripts.

Since our loop model given in figure 5 is the type of nested loop with simple subscript, here the data dependence is considered separately for each individual loop in the nest. Each loop of this nested loop carries cross-iteration dependences if and only if there exist two integers (i, j) satisfying the system of Diophantine equations given by (7) and the system of inequalities given by (8).

$$f_k(I_k) = g_k(I_k) \Diamond a_{k1}I_k + a_{k2} = b_{k1}I_k + b_{k2} \text{ for } 1 \le k \le n \quad (7)$$

$$l_k \le i \le u_k \text{ and } l_k \le j \le u_k \quad (8)$$

And each component of the distance vector, $d_k(i)$ depending on i, as given by(9), has a positive value, then there exists a flow dependence, and if $d_{ak}(j)$ depending on j, as given by(10), has a positive value, then there exists an anti-dependence. Next, if(x, x) is a solution to (7) (x may not be an integer.), then $d_k(x) = d_{ak}(x) = 0$ and there may exist a flow (or anti-) and an anti-dependence (or flow) before and after $I = \lceil x \rceil$, and if x is an integer, then there exists a loop-independent dependence at $I = x$.

$$d_k(i) = j - i = D_k(i)/b_{k1},\ D_k(i) = (a_{k1} - b_{k1})*i + (a_{k2} - b_{k2}) \quad (9)$$

$$d_k(j) = j - i = D_{ak}(j)/a_{k1},\ D_{ak}(j) = (b_{k1} - a_{k1})*i + (b_{k2} - a_{k2}) \quad (10)$$

We can briefly present our proposed method as follows. First, using the procedures in section 4.1, the number of blocks which can be splitted form iteration space is computed for each loop in the nest starting with outermost loop. Next, kth loop which has minimum number of blocks in the nested loop, and the kth and the outermost loops (L_k and L_1) are interchanged for maximizing parallelism available in the loop [10]. Then the outermost loop interchanged (old L_k) is blocked, and all loop nested inside the outermost loop, i.e., from the second loop the nth loop, are transformed to DOALL's. Even if only a loop in the nest does not have dependence, all loops can be transformed to DOALL's.

We can consider the proposed method in two cases: one is that one type of dependence (flow or anti-dependence) exists in the loop and the other is that both flow and anti-dependence exist in the loop.

Here, the number of blocks B_k for each loop in the nest can be computed by Procedure Compute_NB. When there exists a loop-independent dependence in the kth loop.

Procedure Compute_NB

```
/* Computation of the number of blocks for each loop in the nested loop */
BEGIN
  k = 1 ;
  While k ≤ n Do
    If (ak1 = bk1 = 0) then {Bk = 1 ; Fk = 0} ;
    Orif (ak1 = 0, bk1 ≠ 0 or ak1 ≠ 0, bk = 0) then {
      If (lk ≤ i ≤ uk where i = (bk2 – ak2)/ak1 (if ak1 ≠ 0) or
                                (ak2 – bk2)/ak1 (if bk1 ≠ 0))
      then Bk = 3 else Bk = 2; Fk = 1};
    Orif (ak1 = bk1 ≠ 0) then{
      If(ak2 = bk2) then Bk = 1
      else Bk = ⌈(uk – lk)/(ak2 – bk2)/bk1⌉ (if (ak2 - bk2)/bk1 > 0) or
           ⌈(uk – lk)/(bk2 – ak2)/ak1⌉ (if (bk2 - ak2)/ak1 > 0); Fk = 0};
    Orif (ak1*bk1 < 0) then { Bk = 2; Fk = 0};
    Orif (∃ only a flow or anti-dependence in the kth loop) then {
       Compute Bk by step 1-4 in Procedure MaxSplit; Fk = 0 }
    else {Compute Bk by step 5-7 in Procedure MaxSplit_2; Fk = 1}
      k = k+1;
  Endwhile
```

END Compute_NB

First, in case that one type of dependence (flow or anti-dependence) exists in each loop of a nested loop with simple subscripts, Procedure MaxSplit_3 can transform a nested loop into partial parallel loops. As an example, let's consider the loop shown in figure 6. There is one type of dependence exists in each loop of this nested loop. The number of blocks of L_2 is 4 and one of L_3 is 3. Hence, L_3 is interchanged with the outermost loop L_1 for maximizing parallelism. Figure 7 illustrates the result of applying Procedure MaxSplit_3 to the loop in figure 6.

Procedure MaxSplit_3

/* Transformation of nested loops with simple subscripts into partial parallel loops */

BEGIN

Step 1: Compute the number of blocks B_k by Procedure Compute_NB for each loop L_k such that the dependence distance $d_k > 1$ for $1 \le k \le n$;

Step 2: Find L_i such that $B_i = \min(B_k)$ for $1 \le k \le n$;

Step 3: If $i > 1$ then interchange the first loop L_1 with the ith loop L_i ;

Step 4: If the dependence distance of the outermost loop is constant

then split the outermost loop into partial parallel loops by the reduction factor λ (assuming the outermost loop is L_i, $\lambda = \lceil (a_{i2} - b_{i2})/b_{i1} \rceil$ (if $d_i(i) > 0$) or $\lceil (b_{i2} - a_{i2})/a_{i1} \rceil$ (if $d_{ai}(j) > 0$))

else split the outermost loop by Procedure MaxSplit_1;

Transform all loops nested inside the outermost loop, i.e., from the second loop to the nth loop, to DOALL's;

End MaxSplit_3

```
DO I1 = 1, 20
  DO I2 = 1, 20
    DO I3 = 1, 20
S1:     A(I1, 3I2+1, 2I3) = ···
S2:       ··· = A(I1, 2I2-4, I3-3)
    ENDDO
  ENDDO
ENDDO
```

Fig. 6 An example of nested loop with simple subscripts.

```
DOALL I3 = 1, 4
  DOALL I2 = 1, 20
    DOALL I1 = 1, 20
      S1 ;
      S2 ;
    ...
DOALL I3 = 5, 12
  DOALL I2 = 1, 20
    DOALL I1 = 1, 20
    ...
DOALL I3 = 13, 20
  DOALL I2 = 1, 20
    DOALL I1 = 1, 20
```

Fig. 7 The result of the loop in Fig. 6 transformed by Procedure MaxSplit_3.

5. CONCLUSIONS

In this paper, we have studied the parallelization of single loop with non-uniform dependences and nested loops with simple subscripts for maximizing parallelism. For single loops, we can review two partitioning techniques which are fixed partitioning with minimum distance and variable partitioning with ceil(d*(i)*). However, these leave some parallelism unexploited, and the second case has some constraints. Therefore, we proposed a generalized and optimal method to make the iteration space of a loop into partitions with variable sizes (MPSL). By the extension of the MPSL method, we applied to exploit parallelism from nested loops with simple subscripts, based on cycle shrinking and loop interchanging method.

Our future research work is to consider the extension of our proposed method to n-dimensional space.

6. REFERENCES

[1] W. Zhang, G. Chen, M. Kandeemir, and M. Karakoy, "Interprocedural Optimizations for Improving Data Cache Performance of Array-Intensive Embedded Applications," in *DAC 2003, Anaheim*, California, 2003.

[2] D. S. Park, M. H. Choi, "Interprocedural Transformations for Parallel Computing," in *Journal of Korean Multimedia Society, vol. 9, no. 12*, pp.1700-1708, Dec., 2006.

[3] C. D. Ploychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on architecture design," in *IEEE Trans. computers, vol. 37, no. 8*, pp. 991-1004, Aug. 1988.

[4] D. E. Knuth, The Art of Computer Programming, vol. 2: Seminumerical Algorithms, Reading, MA: *Addison-Wesley*, 1981.

[5] H. Zima and B. Chapman, Supercompilers for Parallel and Vector Computers, *Addison-Wesley*, 1991.

[6] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," in *Proceedings of the IEEE, vol. 81, no. 2*, pp.211-243, Feb 1993.

[7] M. J. Wolfe, Optimizing Supercompilers for Supercomputers, Cambridge, *MA: MIT Press*, 1989.

[8] S. J. Jeong, "A Loop Transformation for Parallelism from Single Loops," in *International Journal of Contents, vol. 2, No. 4*, pp.8-11, Dec. 2006.

[9] W. Shang, T. O'Keefe, and J. A. B. Fortes, "On loop transformations for generalized cycle shrinking," in *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 2, pp. 193-204, Feb. 1994.

[10] M. E. Wolfe, and M. S. Lam, "A loop transformation theory and algorithm to maximize parallelism," in *IEEE Trans. Parallel and Distributed Systems, vol. 2, no. 4*, pp. 452-471, Oct. 1991.

Sam-Jin Jeong

He received the B.S. in polymer science from KyungBuk National university, Korea in 1979, and the M.S. in computer science from Indiana university, USA in 1987, and also received Ph.D. in computer science from ChungNam National university, Korea in 2000. From 1988 to 1991, he was a senior research staff at SamSung Electric Co. From 1992 to 1997, he was an assistant professor at HaeCheon University. Since then, he has been with BaekSeok University as a professor. His main research interests include parallelizing compiler, parallel systems, general compiler, and programming languages.