Compiler Techniques to Extract Parallelism within a Nested Loop

Chien-Min Wang and Sheng-De Wang
Department of Electrical Engineering
National Taiwan University
Taipei 10764, Taiwan

Abstract

By analyzing the dependences between instances, we propose a new compiler technique, called cycle breaking, to parallelize nested loops. For a single dependence cycle, it extracts more parallelism than two similar techniques. Several versions of cycle breaking are presented to extract parallelism within a nested loop by linearizing its multi-dimensional iteration space. It is observed that the order in which loops are linearized can dramatically affect the parallelism extracted by cycle breaking. Two loop reordering transformations are investigated. Methods to find the optimal linearization order of loops are proposed. These techniques can enhance the parallelism of a nested loop.

1. Introduction

In this paper, we discuss compiler techniques to extract parallelism within a nested loop. A nested loop is defined as a list of n statements enclosed by m DO loops. The *degree* of a statement is the number of distinct DO loops enclosing it. Let S_i denote the ith statement and I_j denote the index of the jth loop. We use $S_i(I_1, I_2, \ldots, I_k)$ to denote statement S_i of degree k. Each execution of a statement is an *instance* of that statement. Unless explicitly specified, loops are assumed to be normalized. Let N_j be the loop bound of the jth loop. Then, $S_i(I_1, I_2, \ldots, I_k)$ has $\prod_{j=1}^{k} N_j$ different instances. Each instance can be uniquely represented by $S_i(i_1, i_2, \ldots, i_k)$, where i_j is the value of loop index I_j.

To extract parallelism within a nested loop, compilers must know what can be executed in parallel. We use a data dependence graph [2], [4] to represent precedence relations between instances. S_j is dependent on S_i if there exists a data dependence between $S_i(i_1, \ldots, i_k)$ and $S_j(j_1, \ldots, j_k)$ such that $S_j(j_1, \ldots, j_k)$ can not start execution before $S_i(i_1, \ldots, i_k)$ finished execution. $S_i(i_1, \ldots, i_k)$ is called the dependence *source* and $S_j(j_1, \ldots, j_k)$ is called the dependence *sink*. We use $S_i \delta S_j$ to denote the dependence of S_j on S_i.

For each data dependence $S_i \delta S_j$ involving instances $S_i(i_1, \ldots, i_k)$ and $S_j(j_1, \ldots, j_k)$, the rth *distance* ϕ_r, is defined to be $\phi_r = j_r - i_r$. The k-tuple $(\phi_1, \ldots, \phi_k)$ is called the dependence distance vector. Since dependences may exist between several different pairs of instances, the distance vector between two statements may not be unique. The one with the minimum distance should be used as the distance vector. Furthermore, it is assumed that each distance vector is known at compile-time. The true distance or simply distance Φ is defined as follows.

$$\Phi = \sum_{r=1}^{k} \left(\phi_r \prod_{m=r+1}^{k} N_m \right) \quad (1)$$

The true distance gives the total number of iterations between the source and the sink of a dependence. A data dependence graph is a directed graph $G(V, E)$ with a set of nodes $V = \{S_1, S_2, \ldots, S_n\}$ corresponding to the statements in a program, and a set of arcs $E = \{e_{ij} = (S_i, S_j) \mid S_i, S_j \in V\}$ representing data dependences between statements.

The rest of this paper is organized as follows. Section 2 discusses cycle breaking for a single dependence cycle. Section 3 addresses the effect of loop reordering. Section 4 discusses cycle breaking for general dependence graphs. Finally, conclusions are given in section 5.

2. Cycle breaking

Based on the data dependence graph, several different transformations were proposed to restructure a serial program into parallel form. In this section, we propose a new transformation called *cycle breaking*. The serial program is assumed to have n statements involved in a dependence cycle such that $S_1 \delta_1 S_2 \delta_2 \ldots S_n \delta_n S_1$. In the restructured program, parallelism is explicitly specified. We shall use DOALL loops to represent those loops whose iterations can be executed in parallel and DOSER loops to represent those loops whose iterations must be executed serially.

2.1. Simple loops

First, consider the simple loop in Fig. 1(a), which was given as an example in [6]. The dependences between instances of this loop are shown in Fig. 1(b). The instances

0730/3157/91/0000/0024/$01.00 © 1991 IEEE

in Fig. 1(b) are partitioned into disjoint sets of instances called *groups*. Note that instances in the same group can be executed concurrently without violating any dependence constraint. Furthermore, there exists an order of all groups such that the instances of a group are free of dependence sinks when all groups that precede this group have been executed. Groups can be executed serially in this order without violating any dependence constraint. Such an execution order is called a *valid* execution order. Accordingly, the original loop can be restructured into the nested loop in Fig. 1(c) such that 5 instances can be executed in parallel without violating the semantics of the original loop.

This example can be generalized to any simple loop. In general, a simple loop can be represented by Fig. 2(a). Recall that statements are involved in the dependence cycle $S_1\ \delta_1\ S_2\ \delta_2\ \ldots\ S_n\ \delta_n\ S_1$. Let ϕ^i be the distance of the dependence δ_i. The distance of the dependence cycle, denoted by β, is given by the following equation.

$$\beta = \sum_{i=1}^{n} \phi^i \qquad (2)$$

If we traverse from an instance of S_i along dependence arcs until another instance of S_i is reached, we obtain a cycle starting and ending at instances of S_i. Since there is only one dependence cycle, the distance of the path from the starting instance to the ending instance must be β. In other words, there may exist a dependence for more than β consecutive instances of a statement. On the other hand, β consecutive instances of a statement can be executed concurrently without violating any dependence constraint. To extract as more parallelism as possible, cycle breaking will partition instances into groups containing β consecutive instances. The group j of statement S_i, denoted by V_i^j, is the set of instances $S_i((j-1)\beta+\mu_i+1)$ through $S_i(j\beta+\mu_i)$, where μ_i is the offset of S_i and is given by the following equation.

$$\mu_1 = 0 \text{ and } \mu_i = \sum_{j=1}^{i-1} \phi^j \text{ for } i > 1.$$

In other words, $V_i^j = \{\, S_i(k) \mid (j-1)\beta+\mu_i+1 \le k \le j\beta+\mu_i \,\}$. For cycle breaking, the following lemmas hold.

Lemma 1: For any group V_i^j, the instances in V_i^j can be executed in parallel and in any order without violating any dependence constraint.

Lemma 2: For any group V_i^j, the instances in V_i^j are free of dependence sinks if groups are executed as follows.

1. Group V_1^0 is executed first.
2. If $i \neq n$, V_{i+1}^j is executed immediately after V_i^j.
3. V_1^{j+1} is executed immediately after V_n^j.

Proof: First, consider group V_1^0. By definition, group V_1^0 contains no instance. Hence, this lemma is true for V_1^0.

Then, consider group V_{i+1}^j, $i \neq n$. Suppose that $S_{i+1}(k)$ is an instance of group V_{i+1}^j. If $S_{i+1}(k)$ is a dependence sink, the corresponding dependence source must be $S_i(k-\phi^i)$. By definition, $S_i(k-\phi^i)$ is an instance of V_i^j. The instances in V_{i+1}^j are free of dependence sinks if V_i^j has been executed. Therefore, this lemma is true for V_{i+1}^j when $i \neq n$.

Next, consider group V_1^{j+1}. Suppose that $S_1(k)$ is an instance of group V_1^{j+1}. If $S_1(k)$ is a dependence sink, the corresponding dependence source must be $S_n(k-\phi^n)$. By definition, $S_n(k-\phi^n)$ is an instance of V_n^j. The instances in V_1^{j+1} are free of dependence sinks if V_n^j has been executed. Therefore, this lemma is true for V_1^{j+1}.

Since any group must be one of the above three cases, this lemma is true for any group. ■

Lemma 1 indicates that instances of a group can be executed in parallel while Lemma 2 ensures that there exists a valid execution order of groups. Accordingly, any simple loop in Fig. 2(a) can be restructured into the nested loop in Fig. 2(b) without violating any dependence constraint.

2.2. Compare with other techniques

In its purpose, cycle breaking is similar to *partial loop partition* [5] and *cycle shrinking* [6]. We briefly outline these two transformations and then compare cycle breaking with these two transformations. Consider the simple loop in Fig. 2(a). Partition transforms this loop into the nested loop in Fig. 2(c), where $g = \gcd(\phi^1, \phi^2, \ldots, \phi^n)$. It tries to group together all iterations of a DO loop that form a dependence chain. Each such group is executed serially while different groups can be executed in parallel.

On the other hand, cycle breaking will transform the simple loop into the nested loop shown in Fig. 2(d), where $\lambda = \min(\phi^1, \phi^2, \ldots, \phi^n)$. It groups together independent iterations and executes them in parallel. Since $\phi^1, \ldots, \phi^n$ are positive integers, $\min(\phi^1, \phi^2, \ldots, \phi^n) \ge \gcd(\phi^1, \phi^2, \ldots, \phi^n)$ or $\lambda \ge g$. Therefore, the parallelism extracted by cycle shrinking is always greater than or equal to the parallelism extracted by partition. In addition, all n statements inside each iteration can also be executed in parallel. This gives another speedup factor of n.

In contrast to partial loop partition and cycle shrinking, cycle breaking deals with instances rather than iterations. It groups together independent instances of a statement and executes them in parallel. The simple loop in Fig. 2(a) will be transformed into the nested loop in Fig. 2(b), where β is given in Eq. (2). Clearly, we have the following lemma.

Lemma 3: If $\phi^1, \phi^2, \ldots, \phi^n$ are all positive integers, then

$$\beta = \sum_{i=1}^{n} \phi^i \ge n \times \min(\phi^1, \phi^2, \ldots, \phi^n) = n \times \lambda.$$

Lemma 3 indicates that the parallelism extracted by cycle breaking is greater than or equal to the parallelism extracted by cycle shrinking. Hence, for a single dependence cycle, cycle breaking is always superior to partition and cycle shrinking.

2.3. Nested loops

Next, consider applying cycle breaking to nested loops. Let $<\phi_1^i, \phi_2^i, \ldots, \phi_m^i>$ and Φ_i be the distance vector and the true distance of dependence δ_i. For nested loops, there are three versions of cycle breaking as discussed below.

The first version is *simple cycle breaking*. In this version, the dependence cycle is considered separately for each individual loop in the nested loop. For an individual loop at level k, only the kth elements of distance vectors are considered. We have m different cycles, one for each individual loop. For each individual loop, cycle breaking is applied separately as in the simple loop case. Let β_i be the distance of the ith cycle. It is given by the following equation.

$$\beta_i = \sum_{j=1}^{n} \phi_i^j \tag{3}$$

If the distance of the ith cycle is less than or equal to one, then the ith individual loop will be executed serially. Hence, the speedup of the ith individual loop is the maximum of β_i and 1. The total speedup obtained by simple cycle breaking is the product of the speedups of individual loops.

The second version is *true dependence cycle breaking* or *TD cycle breaking*. In this version, only true distances are used. Each dependence arc is labeled by its true distance computed by Eq. (1). Cycle breaking is applied to the nested loop as if it is a simple loop. The true distance of the dependence cycle is computed as follows.

$$\beta_T = \sum_{i=1}^{m} \left(\beta_i \prod_{j=i+1}^{m} N_j \right) \tag{4}$$

In this case, the multidimensional iteration space of the nested loop is treated as a linear space. TD cycle breaking partitions instances of the linearized iteration space into groups containing β_T consecutive instances. The speedup obtained by TD cycle breaking is β_T.

The last version is *selective cycle breaking*. In this version, we consider each element of the distance vectors separately as in the case of simple cycle breaking. Each dependence in a cycle is labeled with the corresponding element of its distance vector. Selective cycle breaking computes the cycle distance β_i for each loop i in the nested loop starting from the outermost loop. This process stops when there exists some j, $1 \le j \le k$, such that $\beta_j \ge 1$. Then, the jth loop is blocked by a factor of β_j. In addition, all loops nested inside the jth loop are transformed into DOALL loops. The speedup obtained by selective cycle breaking is given by the following equation.

$$\beta_S = \beta_j \prod_{i=j+1}^{m} N_i \tag{5}$$

For example, consider the loop in Fig. 3(a). Two statements are involved in the dependence cycle $S_1\ \delta_1\ S_2\ \delta_2\ S_1$. The dependence distance vectors are $<\phi_1^1, \phi_2^1> = <2,4>$ and $<\phi_1^2, \phi_2^2> = <3,5>$. It will be transformed into the loop in Fig. 3(b) if simple cycle breaking is used. The speedup obtained by simple cycle breaking is 45. If selective cycle breaking is used, it will be transformed into the nested loop in Fig. 3(c) and the speedup will become $5N_2$. If TD cycle breaking is used, it will be transformed into the nested loop in Fig. 3(d) and the speedup will become $5N_2+9$.

3. Loop reordering

Note that both TD cycle breaking and selective cycle breaking are special cases of applying cycle breaking to the linearized iteration space of a nested loop. For these two schemes, the order in which a multidimensional iteration space is linearized can dramatically affect the parallelism extracted by cycle breaking. It is possible to greatly enhance the parallelism extracted from a nested loop by rearranging the order in which its multidimensional iteration space is linearized. Two loop reordering transformations are investigated in this section. They are loop interchange [1] and loop reverse. For TD cycle breaking, methods to determine the optimal execution order of loops will be proposed.

The key characteristic of loop interchange is its ability to change the order in which loops are executed. Interchanging loop i and loop j also interchanges pairwisely the ith elements and the jth elements of distance vectors. Accordingly, the ith element and the jth element of the cycle distance vector is also interchanged. It is possible for TD cycle breaking to extract more parallelism through loop interchange. As an example, consider the loop in Fig. 3(a). If the two loops in Fig. 3(a) are interchanged, the parallelism extracted by TD cycle breaking will become $9N_1+5$. For this example, more parallelism can be extracted through loop interchange if $9N_1+5 > 5N_2+9$. The following theorem provides the sufficient and necessary condition for extracting more parallelism through loop interchange.

Theorem 4: Let loop i and loop $i+1$ form a perfectly nested loop. More parallelism can be extracted by interchanging these two loops if and only if $\frac{\beta_{i+1}}{N_{i+1}-1} > \frac{\beta_i}{N_i-1}$.

Proof: Let β and β' be the parallelism extracted by TD cycle breaking from the original loop and the interchanged loop, respectively. More parallelism can be extracted by interchanging loop i and loop $i+1$ if and only if $\beta'-\beta > 0$. From Eq. (4), we can derive the following equation.

$$\beta' - \beta = (\beta_{i+1}N_i + \beta_i - \beta_i N_{i+1} - \beta_{i+1}) \prod_{j=i+2}^{m} N_j$$

Hence, more parallelism can be extracted if and only if $\beta_{i+1}N_i + \beta_i - \beta_i N_{i+1} - \beta_{i+1} = \beta_{i+1}(N_i - 1) - \beta_i(N_{i+1} - 1) > 0$. This completes the proof of this theorem. ■

According Theorem 4, we can determine the optimal execution order of loops as follows. Let $\beta_i / (N_i - 1)$ be the key of loop i. First, we sort loops according to their keys in descending order. Then, we interchange loops according to their orders in the sorted list. The first loop in the sorted list should be the outermost loop in the interchanged loop while the last loop in the sorted list should be the innermost loop in the interchanged loop. Finally, TD cycle breaking is applied to the interchanged loop.

The key characteristic of loop reverse is its ability to reverse the order in which iterations of an individual loop are executed. This changes the signs of the corresponding elements of distance vectors. In general, reversing loop i changes the signs of the ith elements of the distance vectors. The sign of the ith element of the cycle distance vector is also changed. The parallelism extracted by TD cycle breaking may be enhanced through loop reverse. For example, consider the loop in Fig. 4(a). Its cycle distance vector is <5,-9> and the parallelism extracted by TD cycle breaking is $5N_2$–9. If the innermost loop in Fig. 4(a) is reversed as shown in Fig. 4(b), the cycle distance vector will become <5,9> and the parallelism extracted by TD cycle breaking will become $5N_2$+9.

Let β_i denote the ith element of the cycle distance vector. If $\beta_i < 0$, reversing loop i will enhance the parallelism extracted by TD cycle breaking. On the other hand, if $\beta_i > 0$, reversing loop i will reduce the parallelism extracted by TD cycle breaking. In order to extract as more parallelism as possible, we should reverse those loops whose corresponding element of the cycle distance vector is negative. Then, apply loop interchange to the reversed loop. Finally, apply cycle breaking to the interchanged loop.

4. General dependence graphs

In this section, we discuss cycle breaking for nested loops with general dependence graphs. The problem with general dependence graphs is that there may exist many cycles and each cycle may has its own cycle distance. The solution is to focus our attention on shortest paths. Let $A(i,j)$ be the distance of a dependence arc from an instance of S_i to an instance of S_j and $B(i,j)$ be the distance of a shortest path from an instance of S_i to an instance of S_j. The matrix B can be computed from the matrix A by the ALL PAIRS SHORTEST PATHS algorithm [3]. The time complexity is $O(n^3)$. The offsets are given by the following equation.

$$\mu_1 = 0 \text{ and } \mu_i = B(1,i) \text{ for } i > 1.$$

Let β be the number of consecutive instances in a group. Lemma 1 still holds if the following inequality is satisfied.

$$\beta \le \min_i B(i,i) \quad (6)$$

However, there may not exist a valid execution order of groups for any β satisfying Eq. (6). The following theorems give a sufficient condition that there exists a valid execution order of groups.

Theorem 5: The following execution order of groups is acyclic if $\beta < 1 + \min_{i=1}^{n} B(i,i)/n$.

1. Group V_1^0 is executed first.
2. If $\mu_i + A(i,j) + 1 \le \mu_j + \beta$, V_i^k is executed before V_j^k for any k.
3. V_i^{k-1} is executed before V_j^k for any i, j, and k.

Proof: We shall prove this theorem by contradiction. Suppose that the above execution order is cyclic. Without loss of generality, we may assume that groups V_i^k through V_j^k form a cycle. Then, we have the following inequalities.

$$\mu_i + A(i,i+1) + 1 \le \beta + \mu_{i+1} \quad (7a)$$

$$\mu_{i+1} + A(i+1,i+2) + 1 \le \beta + \mu_{i+2} \quad (7b)$$

$$\cdots$$

$$\mu_{j-1} + A(j-1,j) + 1 \le \beta + \mu_j \quad (7c)$$

$$\mu_j + A(j,i) + 1 \le \beta + \mu_i \quad (7d)$$

Accordingly, we can derive the following inequalities.

$A(i,i+1)+\ldots+A(j-1,j)+A(j,i)+(j-i+1) \le (j-i+1)\beta$.

$B(i,i) + (j-i+1) \le (j-i+1)\beta$.

$\beta \ge 1 + B(i,i)/(j-i+1)$.

However, this contradicts the original assumption. Therefore, the above execution order of groups is acyclic under the original assumption. ■

Theorem 6: For any group V_i^j, the instances in V_i^j are free of dependence sinks if groups are executed according to the execution order in Theorem 5.

Proof: First, consider group V_1^0. By definition, group V_1^0 contains no instance. Hence, this theorem is true for V_1^0.

Next, consider group V_j^k. Let $S_j(r)$ be an instance of V_j^k. We have $(k-1)\beta+\mu_j+1 \le r \le k\beta+\mu_j$. Suppose that $S_j(r)$ is a dependence sink of some dependence δ_{ij}. The corresponding dependence source must be $S_i(r-A(i,j))$. Accordingly, $(k-1)\beta+\mu_j-A(i,j)+1 \le r-A(i,j) \le k\beta+\mu_j-A(i,j)$. Note that, by definition, $\mu_j-A(i,j) \le \mu_i$. There are two possibilities:

1. $S_i(r-A(i,j))$ is an instance of group V_i^l and $l < k$. According to rule 3, V_i^l is executed before V_j^k.
2. $S_i(r-A(i,j))$ in an instance of V_i^k. By definition, $(k-1)\beta+\mu_i+1 \le r-A(i,j) \le k\beta+\mu_j-A(i,j)$. According to rule 2, V_i^k is executed before V_j^k.

From the above discussion, we can conclude that the instances in V_i^k are free of dependence sinks according to the execution order in Theorem 5. Since any group must be one of theses cases, this theorem is true for any group. ■

Theorem 5 and Theorem 6 ensure that there exists a valid execution order of groups if the number of instances in a group is chosen appropriately. From Eq. (7), we know that the existence of a valid execution order of groups depends on the number of instances in a group only. Therefore, the maximum number of instances in a group under the constraint that a valid execution order of groups exists can be determined by a binary search.

5. Conclusions

In contrast to partial loop partition and cycle shrinking, cycle breaking deals with instances rather than iterations. Theoretically, this approach will extract more parallelism. For a single dependence cycle, cycle breaking is always superior to partial loop partition and cycle shrinking. We have proposed several versions of cycle breaking for a nested loop by linearizing its multidimensional iteration space. Methods to determine the optimal linearization order of loops have been proposed. These techniques can enhance the parallelism of a nested loop.

Acknowledgements

This work was supported by National Science Council, Taiwan, under the contract no. NSC 79-0416-E002-02, and by the TeleCommunication Lab., Ministry of Communication, under the contract no. TL-79-021.

References

[1] J. R. Allen, and K. Kennedy, "Automatic loop interchange", *ACM SIGPLAN Notices*, vol. 19, no. 6, pp. 233-246, June 1984.

[2] M. Burke, and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *ACM SIGPLAN Notices*, vol. 21, no. 7, pp. 162-175, July 1986.

[3] E. Horowitz, and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Inc., 1978.

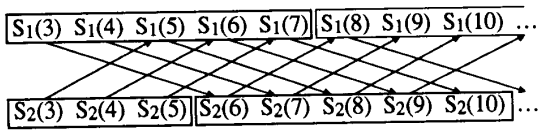
[4] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Proc. Fourth Int. Comput. Software Applications Conf.*, pp. 709-715, Oct. 1980.

[5] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-speed multiprocessors and compilation techniques," *IEEE Trans. Comput.*, vol. C-29, pp. 763-776, Sep. 1980.

[6] C. D. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," *IEEE Trans. Comput.*, vol. C-37, no. 8, pp. 991-1004, Aug. 1988.

```
DO I = 3, N
    S1: A(I) = B(I–2) – 1
    S2: B(I) = A(I–3) * K
ENDDO
```

(a)



(b)

```
DOSER J = -2, N, 5
    DOALL I = max(J,3), min(J+4,N)
        S1: A(I) = B(I-2) - 1
    ENDDO
    DOALL I = max(J+3,3), min(J+7,N)
        S2: B(I) = A(I-3) * K
    ENDDO
ENDDO
```

(c)

Figure 1. An example of cycle breaking.

```
DO I = 1, N
    S1
    ...
    Sn
ENDDO
```

(a)

```
DOSER J = 1–β, N, β
    DOALL I = max(J+μ1,1), min(J+μ1+β–1,N)
        S1
    ENDDO
    ...
    DOALL I = max(J+μn,1), min(J+μn+β–1,N)
        Sn
    ENDDO
ENDDO
```

(b)

```
DOALL J = 1, g
    DOSER I = J, N, g
        S1
        ...
        Sn
    ENDDO
ENDDO
```

(c)

```
DOSER J = 1, N, λ
    DOALL I = J, min(J+λ–1,N)
        S1
        ...
        Sn
    ENDDO
ENDDO
```

(d)

Figure 2. Comparison of cycle breaking with partition and cycle shrinking.

```
DO I = 3, N1
    DO J = 5, N2
        S1: A(I,J) = B(I–3,J–5)
        S2: B(I,J) = A(I–2,J–4)
    ENDDO
ENDDO
```

(a)

```
DOSER K = -2, N1, 5
    DOSER L = -4, N2, 9
        DOALL I = max(K,3), min(K+4,N1)
            DOALL J = max(L,5), min(L+8,N2)
                S1: A(I,J) = B(I–3,J–5)
            ENDDO
        ENDDO
        DOALL I = max(K+2,3), min(K+6,N1)
            DOALL J = max(L+4,5), min(L+12,N2)
                S2: B(I,J) = A(I–2,J–4)
            ENDDO
        ENDDO
    ENDDO
ENDDO
```

(b)

```
DOSER K = -2, N1, 5
    DOALL I = max(K,3), min(K+4,N1)
        DOALL J = 5, N2
            S1: A(I,J) = B(I–3,J–5)
        ENDDO
    ENDDO
    DOALL I = max(K+2,3), min(K+6,N1)
        DOALL J = 5, N2
            S2: B(I,J) = A(I–2,J–4)
        ENDDO
    ENDDO
ENDDO
```

(c)

```
DOSER K = 0–β, (N1–2)*(N2–4)–1, β
    T1 = (K DIV (N2–4)) + 3
    T2 = (K MOD (N2–4)) + 5
    T3 = ((K+β–1) DIV (N2–4)) + 3
    T4 = ((K+β–1) MOD (N2–4)) + 5
    PARBEGIN
        DOALL J = T3, N2
            S1: A(T1,J) = B(T1–3,J–5)
        ENDDO
        DOALL I = T1+1, T2–1
            DOALL J = 5, N2
                S1: A(I,J) = B(I–3,J–5)
            ENDDO
        ENDDO
        DOALL J = 5, T4
            S1: A(T2,J) = B(T2–3,J–5)
        ENDDO
    PAREND
    T5 = ((K+μ2) DIV (N2–4)) + 3
    T6 = ((K+μ2) MOD (N2–4)) + 5
    T7 = ((K+μ2+β–1) DIV (N2–4)) + 3
    T8 = ((K+μ2+β–1) MOD (N2–4)) + 5
    PARBEGIN
        DOALL J = T7, N2
            S2: B(T5,J) = A(T5–2,J–4)
        ENDDO
        DOALL I = T5+1, T6–1
            DOALL J = 5, N2
                S2: B(I,J) = A(I–2,J–4)
            ENDDO
        ENDDO
        DOALL J = 5, T8
            S2: B(T6,J) = A(T6–2,J–4)
        ENDDO
    PAREND
ENDDO
```

(d)

Figure 3. An example of three versions of cycle breaking for nested loops.

```
DO I = 3, N1
    DO J = 5, N2
        S1: A(I,J) = B(I–3,J+5)
        S2: B(I,J) = A(I–2,J+4)
    ENDDO
ENDDO
```

(a)

```
DO I = 3, N1
    DO J = N2, 5, -1
        S1: A(I,J) = B(I–3,J+5)
        S2: B(I,J) = A(I–2,J+4)
    ENDDO
ENDDO
```

(b)

Figure 4. An example of loop reverse.