The Journal of Systems and Software 63 (2002) 91–98

The Journal of Systems and Software

www.elsevier.com/locate/jss

A simple and general approach to parallelize loops with arbitrary control flow and uniform data dependence distances

Weng-Long Chang [a], Chih-Ping Chu [b,*], Jia-Hwa Wu [b]

[a] *Department of Information Management, Southern Taiwan University of Technology, Tainan 701, Taiwan, ROC*
[b] *Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan, ROC*

Received 1 September 2000; received in revised form 1 February 2001; accepted 1 April 2001

Abstract

Loop distribution is applied to exploit the parallelism to loops. For loops with dependence cycle(s) involving all of the statements embedded in control flow, previous methods have significant restrictions. In this paper, an algorithm is proposed to exploit the parallelism for loops under arbitrary control flow and uniform dependence distances. Experiments with benchmark cited from Vector loops, Parallel loops and Livermore loops showed that between 170 subroutines and 24 kernels tested 11 subroutines and 2 kernels had their parallelism exploited by our proposed method.

Keywords: Parallelizing and vectorizing compilers; Data dependence analysis; Control dependence analysis; Loop optimization; Supercomputing

1. Introduction

Loop distribution is applied to exploit the parallelism to loops (Kuck et al., 1981) For loops with control dependences one strategy to parallelize is to convert all IF statements to conditional assignment statements. The resulting loop thus can be distributed by considering only data dependence. This approach, called if-conversion (Zima and Chapman, 1991; Mckinley and Kennedy, 1991; Allen et al., 1983), has been used successfully in a variety of vectorization systems (Zima and Chapman, 1991). However, it has two drawbacks: (1) if vectorization fails, it is not easy to reconstruct efficient branching code and (2) if-conversion may cause significant increases in the code space to hold conditions. For these reasons, research in automatic parallelization has concentrated on an alternative approach that uses control dependences to model control flow. Reconstructing sequential code from a control dependence graph is not trivial, but it is easier than reconstructing from code that has been subject to if-conversion.

Kathryn et al. presented one method to distribute loop in the presence of control flow based on control dependence (Mckinley and Kennedy, 1991). Their method is based on one control dependence graph, G_{cd}, and execution variables having three possible values: true, false, undefined. The execution variables are only needed for branch nodes with at least one successor in a different partition. The execution variables are assigned the value of the test at the branch, capturing the branch decision. Later these variables will be tested to determine control flow in a subsequent partition. Hence, the creation of an execution variable will replace control dependences between partitions with data dependences.

Callahan et al. presented two methods for producing loop distributions in the presence of control flows (Callahan and Kalem, 1987). The first method, which works for structured and unstructured control flow, replicates the control flow of the original loop in each of the new loops by using a data flow graph G_f. Branch variables are inserted to record decisions made in one loop and used in other loops. An additional pass then trims the new loops of any empty control flow. This approach has some of the same drawbacks as if-conversion. The second method, which works only for structured control flow, uses G_f, G_{cd}, and Boolean execution variables. These execution variables indicate that

* Corresponding author. Tel: +886-6-27-57-575x62527; fax: +886-6-27-470-76.
E-mail address: chucp@csie.ncku.edu.tw (C.-P. Chu).

if a particular node in G_f is reached and are created for edges in G_{cd} that cross between partitions. These execution variables are assigned true at the successor indicating that the successor will execute, rather than assigning the decision made at the predecessors. Also, one execution variable may be needed for every successor in the descendent partition. Because their code generation algorithm is based on G_f, rather than G_{cd}. Towle et al. (Baxter and Bauer, 1989; Towle, 1976) use similar approaches for inserting conditional arrays.

Ferrante et al. presented related algorithms whose goals are to avoid replication and branch variables when possible (Ferrante and Mace, 1985; Ferrante et al., 1988). They discuss three transformations that restructure control flow: loop fusion, dead code elimination, and branch deletion. Other research concerned with the definition and use of the program dependence graph does not address distribution. The PTRAN project, which also performs code generation based on G_{cd}, does not address distribution (Cytron et al., 1989; Allen et al., 1987). Work in memory management and name space adjustment (Porterfield, 1989) uses distribution, but only when no control dependences are present. The Stardent compiler (Mckinley and Kennedy, 1991) distributes loops with structured control flow by keeping groups of statements with the same control flow constraints together. For example, all the statements in the true branch of a block IF must stay together, so only the outer level of IF nests can be considered. This limits effectiveness of distribution because partitions are artificially made larger, possibly by grouping parallel statements with sequential ones.

However, the above methods cannot resolve the problem that dependence cycle(s) involve all of the statements in loops embedded in control dependences. In this paper, a method is proposed to exploit the parallelism to loops with dependence cycles and control dependences. The approach is optimal in the sense that it generates the fewest possible predicates. In particular, it introduces one logical array for each conditional node upon which some nodes depend. In addition, the proposed algorithm is shown to generate code for the resulting loop without replicating statements or conditions. In Section 2, we review the concept of data dependence and control dependence. In Section 3, an algorithm for exploitation of parallelism is discussed. In Section 4, an experimental result is given. Finally, in Section 5, conclusions and future work are addressed.

2. Data dependence and control dependence

The statement together with the iteration vector represents an instance of a statement. For example, the instance of a statement S_1 during a iteration vector $\vec{i} = (i_1, \dots, i_n)$ is denoted $S_1(\vec{i})$; the instance of a statement S_2 during a iteration vector $\vec{j} = (j_1, \dots, j_n)$ is denoted $S_2(\vec{j})$. If a statement $S_2(\vec{j})$ uses the array A defined first by another statement $S_1(\vec{i})$, then $S_2(\vec{j})$ is true-dependent on $S_1(\vec{i})$. If a statement $S_2(\vec{j})$ defines the array A used first by another statement $S_1(\vec{i})$, then $S_2(\vec{j})$ is anti-dependent on $S_1(\vec{i})$. If a statement $S_2(\vec{j})$ redefines the array A defined first by another statement $S_1(\vec{i})$, then $S_2(\vec{j})$ is output-dependent on $S_1(\vec{i})$. These data dependences may result in loop-independent and loop-carried data dependence (Allen and Kennedy, 1987) when they exist in statement(s) with indexed variables in loops. Loop-independent dependence refers to the dependence confined within each single iteration, while loop-carried dependence implies the dependence occurring across the iteration boundaries. The loop-carried data dependence can be further distinguished as uniform loop-carried data dependence and non-uniform loop-carried data dependence, depending on whether the dependence is consistent across the loop (Chu and Carver, 1991). In this paper, we do not address non-uniform loop-carried data dependence. The following, Definitions 2.1–2.7, modified or cited from (Banerjee, 1993, 1994; Wolfe, 1996) will be used later.

Definition 2.1. Loop-independent dependence includes loop-independent true-dependence (denoted δ^t), loop-independent anti-dependence (denoted δ^a) and loop-independent output-dependence (denoted δ^o). These relations are represented by the set (denoted Δ), i.e., $\Delta = \{\delta^t, \delta^a, \delta^o\}$.

Definition 2.2. Uniform loop-carried dependence contains uniform loop-carried true-dependence (denoted $[\delta^t]$), uniform loop-carried anti-dependence (denoted $[\delta^a]$) and uniform loop-carried output-dependence (denoted $[\delta^o]$). These relations are represented by the set (denoted $[\Delta]$), i.e., $[\Delta] = \{[\delta^t], [\delta^a], [\delta^o]\}$.

Definition 2.3. The dependence graph of loops is a directed graph where the nodes correspond to the statements in loops, and there is an arc from a node S_1 to another node S_2 iff $S_1\ \delta\ S_2$, where $\delta \in \Delta$ or $\delta \in [\Delta]$.

Definition 2.4. The dependence distance vector from $S_1(\vec{i})$ to $S_2(\vec{j})$, is denoted by $\mathrm{dist}(\vec{i}, \vec{j}) = (j_1 - i_1, \dots, j_n - i_n)$.

Definition 2.5. The dependence distance matrix of nested loops is a matrix whose columns are the distance vectors of all the dependences in nested loops.

Definition 2.6. S_1 is post-dominated by S_2 in a control flow graph G_{cf} if every path from S_1 to the exit node of G_{cf} contains S_2.

Definition 2.7. Given two statements S_1 and S_2 in a control flow graph G_{cf}, S_2 is control dependent on S_1 if and only if:

1. $\exists$ a non-null path P, $S_1 \rightarrow S_2$, post-dominates every node between S_1 and S_2 on P, and
2. S_2 does not post-dominate S_1.

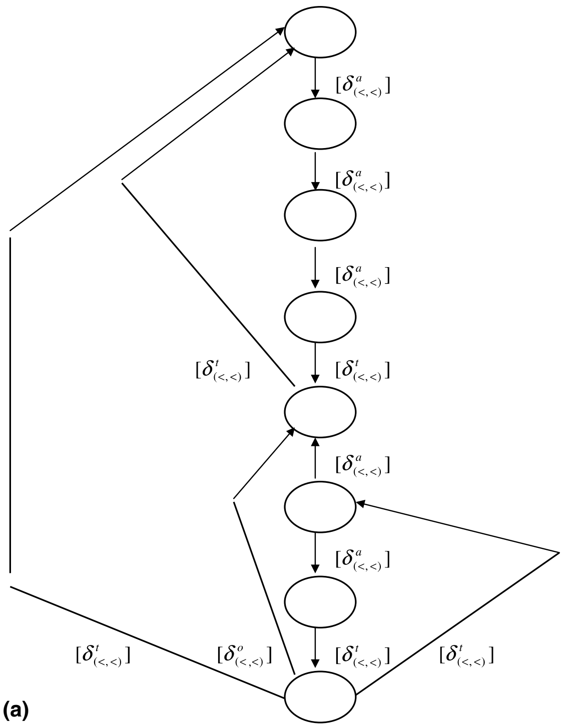
Based on Definitions 2.6 and 2.7, a control dependence graph G_{cd} can be built with the control dependence edges $(S_1, S_2)_l$, where l is the label of the first edge on path $S_1 \rightarrow S_2$ (Ferrante et al., 1987). Intuitively, control dependence between two statements, S_1 and S_2, indicates that the source statement S_1 in the control dependence directly determines whether the sink statement S_2 will execute.

3. Exploitation of parallelism

Consider the loop in Fig. 1. The corresponding data and control dependence graphs for the loop are shown in Fig. 2. The data dependence graph in Fig. 2(a) shows two main dependence cycles. One dependence cycle includes five statements: S_1, S_2, S_3, S_4 and S_5. Another dependence cycle consists of three statements: S_6, S_7 and S_8. Because the loop simultaneously owns control dependences and data dependence cycles, previous methods cannot make the loop executable in either parallel mode or vector mode. In this paper, we propose an algorithm, which can handle this condition. Our proposed method is separated into a three-stage processing: (1) the maximum independent iterations are determined from loop iterations, (2) the corresponding execution variable for each branch node is produced and (3) an equivalent loop is generated based on the control dependence graph. It is clear that the distinct instances of all of the statements in maximum independent iterations determined from loop iterations do not form dependence relations. This is to say that a loop-carried dependence for statements does not exist in the maximum independent iterations. Therefore, the execution of any statement in a control dependence graph may be determined solely from the execution of its predecessor. Execution variables are applied to compute and store decisions for branch nodes. If control dependence successors need only to test the value of the branch for their predecessors, then it is more efficient to run in parallel/vector machine that nested if-statements are transformed into many single if-statements. The details of the proposed algorithm will be described in Section 3.2.

```
DO I=1,N
    DO J=1,N
S1:     IF ( A1(I+2,J+2) .NE. B1(I,J) )THEN
S2:         A1(I+1,J+1)=C1(I+1,J+1)+I+J
S3:         C1(I,J)=D1(I+2,J+2)+J-I
        ELSE
S4:         D1(I+1,J+1)=E1(I,J)+I+J
S5:         B1(I+1,J+1)=D1(I,J)+J
S6:         IF ( F1(I+2,J+2) .NE. B1(I+2,J+2) ) THEN
S7:             F1(I+1,J+1)=E1(I,J)+I
            ELSE
S8:             B1(I+3,J+3)=F1(I,J)+E1(I,J)
            ENDIF
        ENDIF
    ENDDO
ENDDO
```

Fig. 1. An example of loop with data and control dependences.



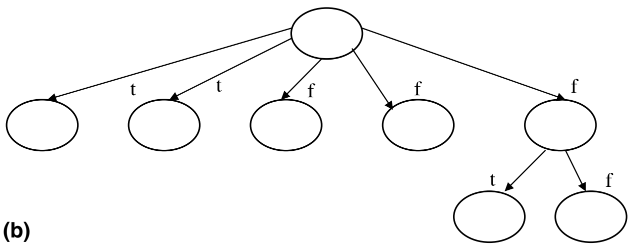


Fig. 2. The data and control dependence G_{cd} graphs for the code shown in Fig. 1.

3.1. Basic concept

Any node in a control dependence graph that has a successor must be a branch node. Because branch nodes correspond to control decisions in the original program, execution variables are only needed for branch nodes. The execution variable is allocated to the value of the test at the branch, capturing the branch decision. Execution variables will be tested to determine control flow. The creation of an execution variable will thus replace control dependences. Execution variables are logical arrays with one value to each iteration of loops, because each iteration can give rise to a different control decision. The following Definition, cited from (Mckinley and Kennedy, 1991) is required for further explanation.

Definition 3.1. Let EV denote an execution variable. An execution variable EV has three possible values: true, false and Ψ, where Ψ denotes an undefined value.

For each branch node a unique execution variable is generated. Because the condition of the first branch node is evaluated at execution time, the corresponding execution variable will not be assigned to an undefined value. For each of the other branch nodes the execution variable is given in advance an undefined value at the outside of loops, indicating that the branch has not yet been executed. If there exists a "not executed" value, then control dependence successors need only to test the value of the execution variable for their specific predecessors. They do not need to test the value of the execution variable to their control dependence ancestors in the entire path.

3.2. The algorithm

Each of the maximum independent iterations determined from loop iterations, as is a task, can execute in parallel/vector machines. The instance of every statement in a task does not form dependence relations. If loop-independent dependences do not exist for the same instance of every statement in maximum independent iterations, then Breadth First Search is applied to trim statements' orders and generate the minimum number of assignments of execution variables to tests of nodes and the minimum number of guards on the values of an execution variable required to correctly determine execution. Otherwise, Depth First Search is used to preserve statement' orders and generate the minimum number of execution variables and guards under such a constraint.

We extend the algorithm in (Mckinley and Kennedy, 1991) to generate the minimal number of execution variables and to produce codes for the resulting loop without replicating statements or conditions. The extended algorithm mainly includes three functions: the parallelizable-loop detector, the execution variable generator and the loop transformer. The task of the parallelizable-loop detector principally figures out maximum independent iterations, MII, from loop iterations. It is to compute minimal dependence distance of each common loop from one dependent distance matrix, $D_{n\times p}$, where p is the number of dependence relations and n is the number of common loops. It is assumed that the number of iteration for the ith common loop is M_i, where $1 \leqslant i \leqslant n$. Let B_i and C_i, respectively, denote independent iteration and minimal dependence distance of each common loop, where $1 \leqslant i \leqslant n$. It is at once derived that independent iteration of each common loop, B_i, is equal to $C_i * \prod_{k=1}^{n} M_k$, where $k \neq i$. Maximum independent iteration is determined from B_i. If the maximum value is equal to zero, then the loop cannot be parallelized and will be preserved. Otherwise, the execution variable generator and the loop transformer will be invoked.

The execution variable generator is mainly used to generate execution variables and assign an initial Boolean value (*true*, *false* or *undefined*) to each execution variable. This function applies Depth First Search to traverse the control dependence graph G_{cd}. It is normally divided into two parts. First, for each of DO-statements in loops the corresponding DOALL-statement with the original index bounds is created and the initial conditions for Depth First Search are set. Then, all of the branch nodes will be found by traversing G_{cd}. For each branch node a unique execution variable is generated. Because the condition of the first branch node is evaluated at execution time, the corresponding execution variable will not be given an initial value. For each of the other branch nodes the execution variable will be assigned an undefined value.

The loop transformer is applied to produce code for the resulting loops without replicating statements or conditions. It involves three phases to generate the restructured loop. At the first phase, a parallelizable-code for DO-statements in loops is produced, its initial conditions for the code generator are set and an assignment of the execution variable to a test of the root in G_{cd} is created. At the second phase, the features of control dependences are determined between a branch node n and its suns. If the left sibling for its sun is one branch node, then the number of if-statements is added one, and its conditional branch is changed to test the execution variable (this is that IF-THEN is generated, where the tested condition is its branch). If there is a true branch between n and its suns and the true test of the execution variable to n has not yet been produced, then its true branch is changed to test the execution variable (this is that IF-THEN is generated, where the condition is its true branch). If there is one *false* edge between n and its suns and the false branch of the corresponding execution variable to n has not yet been produced, then ELSE-IF-THEN is generated, where the condition is its *false* branch. At the third phase, it is determined whether each of the successors to n is a branch node. If it is, then an assignment of the execution variable for a test of the successor is inserted in the restructuring loop. Otherwise, the original statements will be preserved. Repeat phases 2 and 3 until all of the nodes in G_{cd} have been processed. The proposed algorithm and its time complexity are explained as follows.

The parallelizable-loop detector contains four main steps applied to figure out maximum independent iterations. The time complexity to the four steps is $O(p*n), O(n), O(n)$, and $O(y)$, respectively, where p, n and y are, subsequently, the number of the row (i.e., the number of dependence relations), the number of the column (i.e., the number of common loops), and one *constant* value. Therefore, it is at once concluded that the time complexity to the parallelizable-loop detector is $O(p*n+2*n+y)$. A control dependence graph denoted in general is a tree. The feature of a tree will make the processing of traversing a control dependence graph to become simple. The time complexity to Depth First Search applied to traverse each node in a control dependence graph is $O(e)$, where e is the number of edges in a control dependence graph. Because e is equal to $k-1$, where k is the number of nodes in a control dependence graph. Hence, its time complexity is also equal to $O(k-1)$. The execution variable generator uses Depth First Search to determine which nodes in a control dependence graph are branch nodes. It is thus concluded that the time complexity of the execution variable generator is $O(k)$. The loop transformer similarly applies Depth First Search to generate an assignment of an execution variable to a test of each of the branch nodes and guards on the values of each execution variable required. It is similarly derived that the time complexity to the loop transformer is $O(k)$. Therefore, the time complexity to the proposed algorithm is right away derived to be $O(p*n+n+k)$.

Consider the loop in Fig. 1. The dependent distance matrix for the loop is

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & 1 \end{pmatrix}_{2*11}.$$

The parallelizable-loop detector is used to figure out the maximum independent iterations not to be zero and indicate that the loop is parallelizable. The execution variable generator is applied to determine two branch nodes S_1 and S_6. Since the branch node S_1 is the root, the initialization of its execution variable is not created. However, the execution variable for the branch node S_6 is assigned an undefined value. The loop transformer is employed to restructure this loop without replicating statements or conditions. An assignment of the execution variable for the test of S_1 is "$EV_1(I,J) = \text{test}$", where test is the branch condition in S_1. The assignment statement is inserted in the restructuring loop. Since (S_1, S_2) and (S_1, S_4) are on the first true edge and false edge, the true test and the false test are generated as "if ($EV_1(I,J)$ .eq. .true.) then" and "else if ($EV_1(I,J)$ .eq. .false.) then", respectively. Similar processing is also applied to S_6, S_7 and S_8. For other nodes their original statements will be preserved. The restructured loop is shown in Fig. 3.

Algorithm: Loop Parallelization

Input: A loop L with a control dependence graph G_{cd} and one dependent distance matrix D_{n*p}, where p is the number of dependence relations and n is the number of common loops.

Output: A restructured loop L^1.

Step 1. Call the parallelizable-loop detector. If the parallelizable-loop detector indicates that L is parallelizable, then the execution variable generator and the loop transformer are, subsequently, invoked to output L^1. Otherwise, the *original* loop, L, is preserved.

Function: The parallelizable-loop detector

Input: The number of iteration to the ith common loop M_i for $1 \leqslant i \leqslant n$.

Output: Decide whether a loop L is parallelizable.

Step 1. The minimal dependence distance for the ith common loop, C_i, is $C_i = \Phi(D_{n\times p}, i)$, where the subroutine $\Phi(D_{n\times p}, i)$ returns the minimal absolute value for the ith common loop for $1 \leqslant i \leqslant n$.

Step 2. The independent iteration for the ith common loop, B_i, is equal to $C_i * \prod_{k=1}^{n} M_k$, where M_i is the

```
DOALL I=1,N,1
    DOALL J=1,N,1
        EV6 (I,J)=Ψ  /*  Ψ  means 'undefined' */
    ENDDO
ENDDO                                          (a)

DO H = 1, N,1
 DOALL I = H, min(H, N), 1
    DOALL J = 1, N,1
        EV1 (I,J) = A1(I+2,J+2) .ne. B1(I,J)
        if ( EV1 (I,J) .eq. .true.) then
            A1(I+1,J+1) = C1(I+1,J+1) +I+J
            C1(I,J) = D1(I+2,J+2) +J-I
        else if ( EV1 (I,J) .eq. .false.) then
            D1(I+1,J+1) = E1(I,J) +I+J
            B1(I+1,J+1) = D1(I,J)+J
             EV6 (I,J)= F1(I+2,J+2) .ne. B1(I+2,J+2)
        endif
        if ( EV6 (I,J) .eq. .true.) then
            F1(I+1,J+1) = E1(I,J) +I
        else if ( EV6 (I,J) .eq. .false.) then
            B1(I+3,J+3) = F1(I,J) +E1(I,J)
        endif
    ENDDO
  ENDDO
ENDDO                                          (b)
```

Fig. 3. Loop transformation for the code shown in Fig. 1. The code produced by: (a) the execution variable generator, (b) the loop transformer.

number of iteration to the ith common loop and $k \neq i$ for $1 \leqslant i \leqslant n$.

Step 3. The maximum independent iteration, *MII*, is equal to $\mathrm{MAX}(B_i)$, where the subroutine MAX( ) returns the maximum value of arguments for $1 \leqslant i \leqslant n$.

Step 4. If *MII* is not equal to zero, then returns that a loop L is parallelizable. Otherwise returns that it must be preserved.

Function: The execution variable generator.

Input: The root, U, in control dependence graph, G_{cd}, and one stack, Ω.

Output: Generate *undefined* values to the execution variables of branch nodes.

Step 1. Set the initial conditions for Depth First Search and push the root, U, into the stack Ω.

Step 2. Terminate the execution variable generator if the stack Ω is empty.

Step 3. If there is one node W in G_{cd} is adjacent to the top element in the stack Ω and the node W is not traversed, then go to Step 5. Otherwise, go to Step 4.

Step 4. Pop the top element from the stack Ω and go to Step 2.

Step 5. Push the node W into the stack Ω. If there is one edge in G_{cd} to connect the nodes W and X, then generate one *undefined* value to the corresponding execution variable. Go to Step 3.

Function: The loop transformer.

Input: A loop L, the root U in G_{cd} and one stack Ω.

Output: A restructured loop L^1.

Step 1. Reorder DO-statements in an *original* loop L to a sequence such that the loop selected with the maximum independent iterations is the outermost loop and the order of other loops is preserved.

Step 2. Generate one *new* DO-statement outside of the outermost loop with the new indexed variable, the same lower and upper bounds as the loop selected with the maximum independent iterations and the *different* increment denoted to be the minimal dependent distance.

Step 3. Let α, β, and γ represent, respectively, the indexed variable and the upper bound of the *new* DO-statement and the minimal dependent distance. Output one *new* DOALL-statement to replace the *original* loop selected with the maximum independent iterations with the same indexed variable and increment as the *original* loop. The lower and upper bounds of the new DOALL-statement are, respectively, α and $\min(\alpha + \gamma - 1, \beta)$, where the subroutine min() returns the smallest value of arguments.

Step 4. Generate DOALL-statements to replace the other loops with the same indexed variable, lower bound, upper bound and increment.

Step 5. Set the number of if-statements to zero, push the root U into the stack Ω and create one assignment of the execution variable to a test of the root.

Step 6. Go to Step 13 if the stack Ω is empty.

Step 7. If there is one node W is adjacent to the top element in the stack Ω and the node W is not traversed, then go to Step 9. Otherwise, go to Step 8.

Step 8. If the top element in the stack Ω is one branch node and the number of if-statements is greater than zero, then generate END-IF and decrease one to the number of if-statements. Pop the top element from the stack Ω and then go to Step 6.

Step 9. Push the node W into the stack Ω. If the left sibling for the node W is one branch node, then the number of if-statements is added one, its conditional branch is changed to test the execution variable (this is that IF-THEN is generated, where the tested condition is its branch), and then go to Step12. Otherwise, go to Step 10.

Step 10. If there is one *true* edge in G_{cd} to connect the nodes V and W and the true branch of the corresponding execution variable has not yet been produced, then the number of if-statements is added one, its true branch is changed to test the execution variable (this is that IF-THEN is generated, where the condition is its true branch), and then go to Step12. Otherwise, go to Step 11.

Step 11. If there is one *false* edge in G_{cd} to connect the nodes V and W and the false branch of the corresponding execution variable has not yet been produced, then ELSE-IF-THEN is generated, where the condition is its *false* branch. Go to Step 12.

Step 12. If there is one edge in G_{cd} to connect the nodes W and X, then an assignment of the execution variable for a test of the node W is generated, END-IF is generated and the number of if-statements is decreased one. Otherwise, the original statement for the node W is preserved. Go to Step 7.

Step 13. Generate the corresponding ENDDO-statement(s).

Step 14. Terminate the loop transformer.

3.3. Optimality

This section proves that the proposed algorithm creates the minimal number of execution variable needed to track control decisions affecting statement execution in the reconstructed loop. It also establishes that the algorithm produces the minimal number of guards on the values of an execution variable required to correctly execute the produced code. Therefore, our algorithm is optimal for loops with arbitrary control flow and uniform dependence distance. The following lemmas is extended from (Mckinley and Kennedy, 1991).

Lemma 3.1. *Each execution variable represents a unique decision that must be communicated between a branch node and its successor(s).*

Proof. When a decision in one statement directly affects the execution of another statement, as specified by G_{cd}, the corresponding execution variable is created. According to the definition of G_{cd}, it is very obvious that no decision node includes another, and hence any decisions represented by execution variables are unique. □

The restructuring algorithm creates the minimal number of guards on the values of an execution variable required to correctly determine execution. Let m be the number of distinct branch labels that contain successors of node n. There are at most k tests on the values of an execution variable EV_n. If all of the $n's$ successors are not branch nodes, then k is possibly one or two. If all of distinct branch labels are the same true branch labels, then k is equal to one. Otherwise, k is equal to two. If all of the n's successors are branch nodes, then k is equal to m. It is concluded that k is bounded by the number of n's successors that are branch nodes. We thus get $1 \leqslant k \leqslant m$.

Lemma 3.2. *The number of guards that test an execution variable is the minimal required to preserve correctness for the proposed code.*

Proof. Using contradiction. Guards would be created that were either unnecessary or redundant if there is a version of the method with fewer guards. Lemma 3.1 would be disturbed if there were redundant guards. If there were unnecessary guards, then would be multiple guards for nodes with the same label. However, the method generated at most one guard per label. □

4. Experimental results

Both the original and transformed versions of the program were tested to evaluate the performance of the proposed scheme. For the test machine, we choose the DECmpp model 12,000 which has 1024 data processors in our environment. Livermore Loop, Parallel Loop and Vector Loop were used as benchmarks (Levine et al., 1991; Dongarra et al., 1991; Arnold, 1982). Some loops in the two benchmarks were modified to become structured loops without go-to statements. The original programs tested in the two benchmarks were executed in scalar mode (sequence mode). The transformed versions of the same original program tested were run in parallel mode.

Suppose that k_{SM} and k_{PM} are the execution time of the original programs and the transformed programs, subsequently. The speed-up in Table 1 is defined to be the set of k_{SM}/k_{PM}. Each row in Table 1 shows how many times the execution time of the original program took longer than the execution time of the transformed program. For example, the first row shows that there was one subroutine, called S277, in which the execution time of the original codes took 61.2 times longer than that of the transformed codes. For all of the subroutines in our experiments, the execution time of the original programs was indicated to take from 1.9 to 61.2 times longer than the execution time of the transformed programs. This indicates that that the proposed scheme is very significant in term of speed-up, ranging from 1.9 to 61.2.

Table 1
The performance of the proposed scheme to loops in two benchmarks

Benchmark	Loop name	Speed-up
Vector loop	S277	61.2
Vector loop	S274	50.3
Vector loop	S273	42.5
Parallel loop	DO_3000	26.3
Parallel loop	DO_3100	24.6
Vector loop	S279	22.4
Parallel loop	DO_3800	20.5
Livermore loop	Kernel 22	18.6
Vector loop	S253	12.3
Parallel loop	DO_3500	10.7
Parallel loop	DO_3600	8.6
Livermore loop	Kernel 15	7.4
Vector loop	S2710	1.9

5. Conclusions and future works

We have presented a very general and optimal algorithm for loops with control flow and *uniform* dependence distances. The algorithm can be used to enhance the effectiveness of vectorizers, parallelizers and programming environments, alike. The future research will focus on discovering dependence distance algorithms that are effective in deciding whether loops with control flow and *non-uniform* dependence distances can be reconstructed.

Acknowledgements

This work was partially supported by the National Science Council of Republic of China under grant NSC89-2213-E-168-013.

References

Allen, F., Burke, M., Charles, P., Cytron, R., Ferrante, J., 1987. An overview of the PTRAN analysis for multiprocessing. In: Proceedings of the First International Conference on Supercomputing.

Allen, R., Kennedy, K., 1987. Automatic translation of fortran program to vector form. ACM Trans. Programming Languages Syst. 9 (4), 491–542.

Allen, R., Kennedy, K., Porterfield, C., Warren, J., 1983. Conversion of control dependence to data dependence. In: Conf. Rec. 10th ACM Symposium Principles of Programming Languages (POPL), pp. 177–189.

Arnold, C.N., 1982. Performance evaluation of three automatic vectorization packages. In: Proceedings of the 1982 International Conference on Parallel Processing, pp. 235–242.

Banerjee, U., 1993. Loop Transformation for Restructuring Compilers: The Foundations. Kluwer Academic Publishers, Dordrecht, ISBN 0-7923-9318-X.

Banerjee, U., 1994. Loop Parallelization. Kluwer Academic Publishers, Dordrecht.

Baxter, W., Bauer, H.R., 1989. The program dependence graph and vectorization. In: Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages, Austin, TX.

Callahan, D., Kalem, M., 1987. Control Dependence Supercomputer Software. Newsletter 15, Department of Computer Science, Rice University.

Chu, C.-P., Carver, D.L., 1991. An analysis of recurrence relation in Fortran Do-loops for vector processing. In: Proceedings of the Fifth Parallel Processing Symposium. IEEE CS Press, Los Alamities, CA, pp. 619–625.

Cytron, R., Ferrante, J., Sarker, V., 1989. Experiences using control dependence in PTRAN. In: Proceeding of the Second Workshop on Languages and Compilers for Parallel Computing.

Dongarra, J., Furtney, M., Reinhardt, S., Russell, J., 1991. Parallel loops – a test suite for parallelizing compilers: description and example results. Parallel Computing 17, 1247–1255.

Ferrante, J., Mace, M., 1985. On linearizing parallel code. In: Conference Record of the 12th Annual ACM Symposium on the Principles of Programming Languages, New ORLEANS, LA.

Ferrante, J., Mace, M., Simons, B., 1988. Generating sequential code from parallel code. In: Proceedings of the Second International Conference on Supercomputing, St. Malo, France.

Ferrante, J., Ottenstein, K., Warren, J., 1987. The program dependence graph and its use in optimization. ACM Trans. Programming Languages Syst. 9 (3), 319–349.

Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M., 1981. Dependence graphs and compile optimizations. In: Conf. Rec. of 8th ACM Symposium on Princ. of Programming Language.

Levine, D., Callahan, D., Dongarra, J., 1991. A comparative study of automatic vectorizing compilers. Parallel Computing 17, 1223–1244.

Mckinley, K., Kennedy, K., 1991. Loop Distribution with Arbitrary Control Flow. International Conference on Super-computing.

Porterfield, A., 1989. Software Methods for improvement of Cache performance. Ph.D. Thesis, Department of Computer Science, Rice University.

Towle, R.A., 1976. Control and data dependence for program transformation. Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

Wolfe, W., 1996. High Performance Computer for Parallel Computing. Addison-Wesley, Reading, MA, ISBN 0-8053-2730-4.

Zima, H., Chapman, B., 1991. Supercompilers for Parallel and Vector Computers. Addision-Wesley, Reading, MA, ISBN 0-201-17560-6.

Weng-Long Chang received his BS degree in computer science and information engineering from Feng China University, Taiwan, in 1988 and MS and PhD degrees in computer Science and information engineering from the National Cheng Kung University, Taiwan, in 1994 and 1998, respectively. He is currently an Assistant Professor in the Department of Information Management of Southern Taiwan University of Technology, Taiwan. His research interests include languages, tools and compilers for parallel computing.

Chih-Ping Chu received a BS degree in agricultural chemistry from National Chung Hsing University, Taiwan, and MS degree in computer science from the University of California, Riverside, and a PhD degree in computer science from Louisiana State University. He is currently a professor in the Department of Computer Science and Information Engineering of National Cheng Kung University, Taiwan. His research interests include parallel computing, parallel processing, component-based software development, and internet computing.

Jia-Hwa Wu received his BS degree in mechanical engineering from Feng Chia University, Taiwan, in 1981 and MBA degree in industrial management from National Cheng Kung University, Taiwan, in 1986. He is currently a doctoral candidate in computer science and information engineering at the National Cheng Kung University, Taiwan. His research interests include parallelizing compilers, data mining, and internet computing.