DEPENDENCE ANALYSIS AND EVALUATION OF INHERENT PARALLELISM OF PROGRAMS

ZHENJIE CHEN

Dependence Analysis and Evaluation of Inherent Parallelism of Programs

by

Zhenjie Chen

A thesis submitted to the School of Graduate Studies

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

December 1995

St. John's Canada

ISBN 0-612-13886-0

Abstract

Dependencies are relations between statements of a program. They indicate the constraints imposed on the order of statement execution, and are often used for the evaluation, optimization, vectorization and parallelization of programs. By means of dependence analysis, much work has been done to exploit the parallelism in the loops in which there are no dependencies that cross from one iteration of the loop to another. Only recently an approach was proposed for exploiting the parallelism available in loops with cross-iteration dependencies.

The aim of this project is to evaluate inherent parallelism of sequential programs by means of dependence analysis. The thesis first introduces the definitions, concepts and basic dependency analysis algorithms, and presents a uniform representation of dependencies called a dependence graph. Then, using the dependence graph, a general approach is presented which can be used to analyze the parallelism between loops as well as between loops and other parts of a program. This approach was implemented as a program called DSA (Dependence and Speedup Analyzer), used to perform the dependence analysis and to evaluate the inherent parallelism of Fortran programs. Finally, the implementation of DSA is briefly described and its use is illustrated by a series of examples.

Acknowledgments

I wish to express my thanks to my supervisor, Prof. Wlodek M. Zuberek, for his guidance, interest, suggestions, patience and financial assistance during my studies at Memorial University of Newfoundland. I learned a lot from our numerous discussions. He has contributed significantly to the quality of this thesis.

I also wish to thank my other instructors: Prof. John Shieh, Prof. Tony Middleton, Prof. Krishnamurthy Vidyasankar, Prof. Caoan Wang and Prof. Xiaobu Yuan.

I am very grateful to the administrative and technical staff who have helped in one way or another in the preparation of this thesis. Special thanks are due to Ms. Elaine Boone, Mr. Michael Rayment, Mr. Nolan White, Mrs. Patricia Murphy, and Mrs. Jennifer Cutler for their help and assistance.

I would like to thank Dr. Siwei Lu, Dr. Jian Tang, Mr. Hao Chen, Ms. Xi Lu, Mrs. Zhengqi Lu, Mr. Donald Craig, Mr. Zhihong Yuan, Mr. Yaguang Chen and Mr. Ming Tan for their valuable comments and suggestions.

I would also like to thank many others not named here who provided encouragement and assistance during my graduate studies.

Finally, I would like to thank the School of Graduate Studies and the Department of Computer Science, for providing, together with Dr. W.M. Zuberek, financial support during my studies.

This thesis is dedicated to

my parents

for their support and encouragement throughout my education

and to

my wife, Bing Mi, and daughter, Xiaoyue Chen,

for enduring the hardship of the past three years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The need for computing power has been growing steadily in the last two decades. However, with the slowing rate of improvements in semiconductor technologies, the processing ability of single-processor systems and sequential processing of programs are reaching their limits. In turn, this has been stimulating research in multiprocessor architectures and parallel algorithms.

Much work in the area of exploiting the parallelism of programs and program conversion from sequential to parallel form has been done on the basis of dependence analysis. Several experimental compiling systems exploiting parallelism in FORTRAN programs have been developed using dependence analysis [2, 3, 7]. However, these systems perform parallelization in a rather limited range, often analyzing only the DO constructs of FORTRAN programs, while the parallelism between other statements is ignored. For example, Gupta and Soffa [18, 19] developed specialized compilation techniques which can be used to detect parallel operations within and between sequential statements; their work was done for the Reconfigurable Long Instruction Word (RLIW) architecture model, and the increased computation speed was obtained by matching an application program to the particular RLIW architecture in structure and in size. In

more general approaches, the evaluation and detection of parallelism should be machine independent [47]. Recently Lilja [32] developed a method called *critical dependence ratio* to determine maximum possible parallelism for a loop, given unlimited hardware resources. However, the method cannot deal with parallelism between loops or between a loop and the other parts of a program.

The aim of this research is to use dependence analysis for evaluation of inherent parallelism of sequential programs. Dependencies are relations between statements of a program. They can be represented as a graph, called *a program dependence graph*, and are widely used for performing program optimizations, vectorization, and parallelization [20, 31, 34, 29, 30, 15]. It has been shown that if the program dependence graphs of two programs are isomorphic, then the programs are strongly equivalent in the sense of their behaviors [23]. This equivalence can be used to determine the maximally parallel execution of the program. Such an approach is the motivation for this work. The results of this research can be used in automatic transformation of sequential programs to their equivalent parallel forms.

There are two main contributions of this research. First, a general approach to finding the maximal possible parallelism of a sequential program is proposed. Secondly, the proposed approach is implemented as a program called DSA (Dependence and Speedup Analyzer). The program performs the dependence analysis and the evaluation of inherent parallelism of FORTRAN programs.

1.1 Basic Concepts

Dependencies arise as the result of two separate effects. First, a dependence exists between two statements S_i and S_j if both statements access the same memory location

(at least one of them must write this location) and no statement between S_i and S_j writes this location. Dependencies of this type are called *data dependencies*[1]. There are three types of data dependencies based upon the ways in which S_i and S_j access the location. Statement S_j is

- *flow-dependent on* S_i, if S_i writes a memory location and S_j reads it;
- *anti-dependent on* S_i, if S_i reads a memory location and S_j writes it;
- *output-dependent on* S_i, if S_i writes a memory location and S_j writes it again;

The memory location can correspond to a scalar variable or an array element.

Secondly, a dependence exists between a statement S and a predicate B whose value (directly) controls the execution of S. Dependencies of this type are called *control dependencies*[2]. For example, in the sequence of statements:

```
S1:  IF (B) THEN
S2:      X=Y+W
S3:      Z=X*A
S4:      A=C-D
S5:      Z=E+F
S6:  ENDIF
```

The statements S_2, S_3, S_4 and S_5 are control-dependent on the predicate B; in other words, S_2, S_3, S_4 and S_5 are control–dependent on S_1. S_3 is flow-dependent on S_2 due to X, S_4 is anti-dependent on S_3 due to A, and S_5 is output-dependent on S_3 due to Z.

Dependence analysis detects the dependencies in a program. Ferrante [15] has made an excellent contribution to the analysis of control dependencies. Data dependence

[1] A formal definition of data dependencies is given in Chapter 2.
[2] A formal definition of control dependencies is given in Chapter 2.

analysis is more complicated than control dependence analysis because it must take into account dependencies created by subscripted variables (array elements), and *aliases*, *i.e.*, references to memory locations which are identified by more than one identifier (aliases can be created by procedure passing mechanisms and data equivalences). For example, in the following programs:

```
S1:  DO I=1,10
S2:       A(I)=B(I)
S3:       C(I)=D(I)
S4:       B(I)=A(I+1)+F(I)
S5:  ENDDO
```

it is easy to see that S_4 is anti-dependent on S_2 due to B(I). However, S_2 is also flow-dependent on S_3 due to A(I) and A(I+1) when the variable I increases in repeated executions of the loop. On the other hand, if C is an alias of B, then S_3 is anti-dependent on S_2, and S_4 is output-dependent on S_3 due to C(I) and B(I).

The *speedup factor*, used to measure the inherent parallelism of programs, is defined as

$$speedup = \frac{T_{serial}}{T_{parallel}},$$

where T_{serial} is the time of the sequential execution of a program, and $T_{parallel}$ is the time of the maximally parallel execution of the same program, *i.e.*, the time of program execution with an unlimited number of available processors. The speedup factor can further be 'specialized' as *fixed size speedup* and *scaled speedup* [45]. Fixed size speedup indicates how much execution time can be reduced on a specific parallel processor, while scaled speedup is used in exploring the computational power of parallel computers for solving otherwise intractable problems.

From the algorithm analysis point of view, the speedup factor is defined as [13] $\frac{E(T_1)}{E(T_p)}$, called *speedup of the average execution times*, or $M(\frac{T^A}{T^B})$, called *average speedup*, where T_1 and T_p are random variables representing the execution time on one and on p processors, respectively, $E(T)$ is the expected value of T, T^A and T^B are random variables representing the execution time of a sequential algorithm A and a parallel algorithm B for solving the same problem, and $M(T)$ can be any mean value of T, in particular the arithmetic mean. It should be noted that these two definitions also provide two methods to calculate the approximate values of the speedup factor.

The standard definition of the speedup factor, i.e., $\frac{T_{serial}}{T_{parallel}}$, is used in this thesis to evaluate the inherent parallelism of programs on the basis of control and data dependencies. Control and data dependencies of a program represent control and data flow relationships which must be respected by any execution of the program, whether parallel or sequential. By examining these dependencies, we can extract the inherent parallelism in a program and evaluate the speedup factor.

As an illustration, the following program can be considered:

```
S1:  DO I=1,10
S2:       C(I)=A(I)-B(I)
S3:       D(I)=A(I)+B(I)
S4:       E(I)=C(I)*C(I)
S5:       F(I)=D(I)*D(I)
S6:       C(I)=E(I)+F(I)
S7:  ENDDO
```

the statements S_2 to S_6 are control-dependent on S_1 since the value of the loop index variable I determines whether S_2 to S_6 are executed. S_4 is flow-dependent on S_2 due to C(I), S_5 is flow-dependent on S_3 due to D(I), S_6 is output-dependent on S_2 due to C(I), S_6 is anti-dependent on S_4 due to C(I), S_6 is flow-dependent on S_4 due to E(I),

and S_6 is flow–dependent on S_5 due to F(I). These dependencies can be represented as a graph, as shown in Figure 1.1.

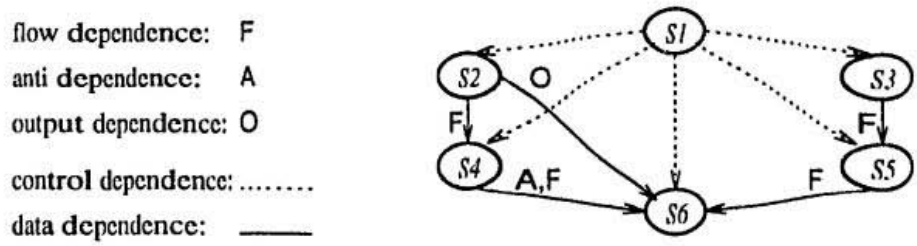


Figure 1.1: Dependence graph.

Assuming that all arithmetic, logical or assignment operation can be completed in one unit of time:

$$\begin{aligned} T_{serial} &= (10+1)*1+10*(t_{S_2}+t_{S_3}+t_{S_4}+t_{S_5}+t_{S_6}) \\ &= 11+10*(2+2+2+2+2) \\ &= 111\ (units). \end{aligned}$$

If the number of available processors is unlimited, the loop can be unfolded into 10 groups, and these 10 groups can be executed in parallel. To find $T_{parallel}$, we need only to consider the time to execute one group as all groups are identical. Since S_2 and S_4 have no dependence relation with S_3 and S_5, and all statements S_2 to S_6 are control–dependent on S_1, then S_2 and S_4 can execute in parallel with S_3 and S_5. S_6 is dependent on S_2, S_4 and S_5, so S_6 must execute after S_2, S_4 and S_5. Therefore:

$$\begin{aligned} T_{parallel} &= \max(t_{S_2}+t_{S_4}, t_{S_3}+t_{S_5})+t_{S_6} \\ &= \max(2+2, 2+2)+2 \\ &= 6\ (units). \end{aligned}$$

So, the speedup factor is 18.5 in this case.

1.2 Thesis Overview

This thesis is organized into six chapters. Chapter 2 reviews the research on dependence analysis and alias analysis. Then, the concepts and algorithms related to control and data dependence analysis as well as alias analysis are introduced in detail. Finally, a uniform representation of dependencies, the *dependence graph*, is presented. Chapter 3 is devoted to the evaluation of inherent parallelism including a brief introduction to the evaluation of T_{serial} and a detailed presentation of a general approach to evaluate $T_{parallel}$. This approach is based on the dependence graph of a program and can be used to deal with the parallelism between loops and between loops and other blocks of the program. Chapter 4 describes the implementation of DSA, a program performing dependence analysis and evaluation of the speedup factor, which uses the algorithms and approach presented in Chapters 2 and 3. Examples and conclusions are presented in Chapters 5 and 6, respectively.

Chapter 2

Dependence Analysis

Research on dependence analysis has been conducted over the last twenty years. Ferrante [15] made an excellent contribution to analysis of control dependencies, characterizing the control structure of programs. Analysis of data dependencies is more difficult than that of control dependencies. It has been shown that the detection of data dependencies among subscripted variables is an NP–complete problem [17, 36]. The first contribution to the detecting of data dependence among subscripted variables is due to Banerjee [4, 5, 6]. He proposed an inequality which provides a sufficient condition for the existence of data dependencies. The Banerjee's inequality decision algorithm can be used to deal with more complicated data dependence testing problems, but it is more complex and inefficient. Another significant result is Allen and Kennedy's GCD decision algorithm [3] derived from the number theory, which also provides a sufficient condition for the existence of data dependencies. The GCD decision algorithm is fast and efficient for some special data dependence cases. Therefore, in practice, the GCD decision algorithm is usually used first. If data independence can be found, then the testing procedure is over. Otherwise, the Banerjee's decision algorithm is used to further perform the data dependence testing. A more practical solution comes from Burke and Cytron's

hierarchical dependence testing algorithm [9]. It is usually used as a test framework and is combined with other testing algorithms, such as the Banerjee and GCD decision algorithms. Due to the inherent intractability of data dependence testing among subscripted variables, research on data dependence is continuing [49, 37, 41, 40, 33].

Another factor which makes data dependence analysis complicated is the existence of aliases created by procedure passing mechanisms and data equivalences. To perform data dependence analysis, alias analysis is needed first. In FORTRAN programs, aliases caused by data equivalences are always declared explicitly by the COMMON and EQUIVALENCE statements, so detection of such aliases is quite straightforward. However, an inter-procedural alias analysis must be performed to find the aliases caused by procedure passing mechanisms. A significant work on inter-procedural alias analysis was first done by Ryder [42]. She introduced a representation, called the *call graph*, of the control and data flow in programs to investigate inter-procedural communication. Burke and Cytron [9] also proposed some methods to identify aliased arrays, and to propagate inter-procedural information. Further improvement is due to Cooper and Kennedy [10, 11], who presented a fast algorithm for computing inter-procedural aliases based on an improved call graph, called the *binding graph*. An improved version of the fast alias analysis algorithm based on binding graph is presented in [38]. The newest result is due to [39].

The following four sections of this chapter discuss the control dependence analysis, data dependence analysis, alias analysis, and dependence graph, respectively.

2.1 Control Dependence Analysis

To simplify the discussion, it is assumed that a program contains only assignments used in the sequence, selection and iteration constructs. The sequence, selection and iteration constructs have the following forms:

- *sequence*: $S_1;S_2$
- *selection*: **if** B **then** S_1 **else** S_2 **endif**
- *iteration*: **for** $i := 1$ **to** n **do** S **enddo**

where B is a boolean expression, n is a constant or a variable, and S, S_1 and S_2 are assignment statements, sequence constructs, selection constructs or iteration constructs. The boolean expression B is called the *branch condition* of the selection construct. The boolean expression $i \leq n$, which is the condition to continue the iteration, is called the *branch condition* of the iteration construct.

The *control flow graph* G of a program is a directed graph $G = (N, E)$, where the set of nodes, N, is the set of assignments and branch conditions of selection and iteration constructs in the program, and the edges, $E \subseteq N \times N$, represent possible transfers of control between nodes. It is assumed in control flow graphs that nodes which represent branch conditions (they always have two immediate successors) have attributes T (*true*) and F (*false*) associated with the outgoing edges. Each control flow graph is augmented with two special nodes: ENTRY and STOP, which represent the unique beginning and termination of program execution. ENTRY has one edge labeled "T" outgoing to the first statement of the program and another edge labeled "F" outgoing to STOP.

The following two definitions were introduced in [15] together with a general idea of analyzing control dependencies.

Definition [15]. Let G be a control flow graph. A node v in G is *post-dominated* by a node w if every directed path from v to STOP (not including v) contains w.

If v is *post-dominated* by w, w is called a *post-dominator* of v. Note that this definition of post-dominance does not include the initial node of the path. In particular, a node never post-dominates itself.

Definition [15]. Let G be a control flow graph. Let x and y be nodes in G. y is *control-dependent on* x iff:

1. there exists a directed path p from x to y with any node z of p (excluding x and y) post–dominated by y, and

2. x is not post-dominated by y.

In other words, if y is control-dependent on x in a control flow graph, then there must exist at least two paths from x to STOP in the graph; one includes y and the other does not.

Definition [15]. Let $G = (N, E)$ be a control flow graph. A *post–dominator tree* $T = (N, E')$ contains the set N of nodes of G, and the subset E' of the edges E of G such that if v is post–dominated by w, or w is a post–dominator of v, there must exist a path from w to v in T.

Definition [15]. Let T be a post–dominator tree, and a and b two nodes in T. A node c of T is called the *common ancestor* of a and b if T contains two paths, one from c to a and the other from c to b. A node l of T is called the *least common ancestor* of a and b if:

- l is a common ancestor of a and b, and

- there is no other l' in T such that l' is also a common ancestor of a and b, and there is a path from l to l' in T.

Given a control flow graph, control dependencies can be determined in the following three steps [15]:

1. Find post-dominators in the control flow graph, and construct the post-dominator tree T.

2. Find a set S which consists of all edges (a, b) in the control flow graph such that there is no path from b to a in T (*i.e.*, b does not post-dominate a). Note that in this case the edge (a, b) must be labeled by "T" or "F".

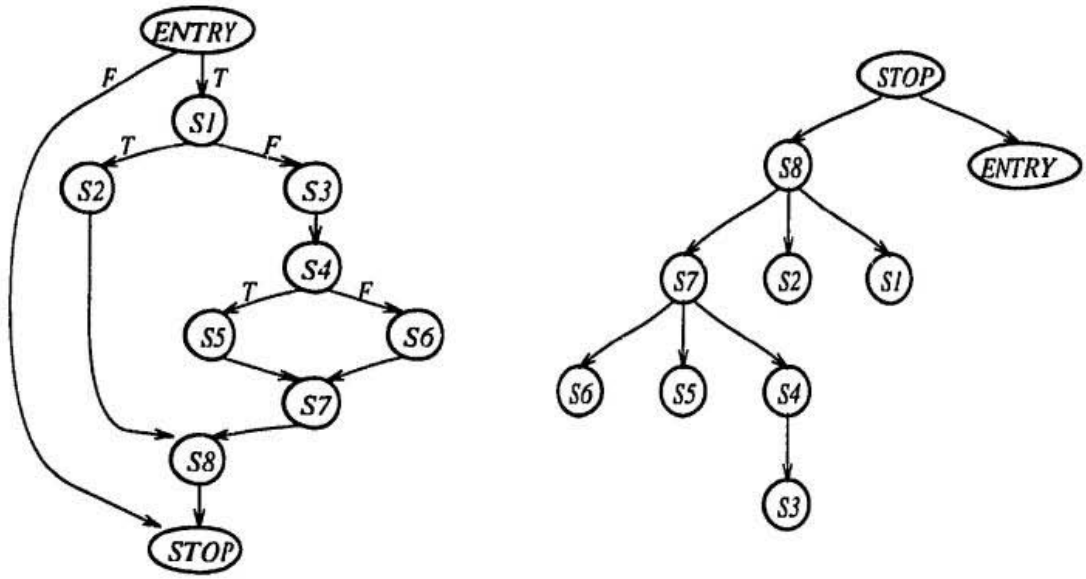
3. For each edge (a, b) in S, find the least common ancestor l of a and b in T. It has been shown [15] that either l is a or l is the parent of a in T.

   - If l is a, all nodes in the post-dominator tree on the path from a to b, including a and b, are control-dependent on a.

   - If l is the parent of a, all nodes in the post-dominator tree on the path from l to b, including b but not l, are control dependent on a.

For example, for the following program:

```
S1:  IF (A) THEN
S2:       Y=X+Z
     ELSE
S3:       P=M-S
S4:       IF (B) THEN
S5:            V=P+S
          ELSE
S6:            U=Y-Z
          ENDIF
```



(a) The control flow graph of the program. (b) The post-dominator tree of the program.

Figure 2.1: The control flow graph and its post–dominator tree.

```
S7:       Q=U
      ENDIF
S8:   T=Y
```

the control flow graph and the post-dominator tree are shown in Figure 2.1. In this example, $S = \{(ENTRY, S1), (S1, S2), (S1, S3), (S4, S5), (S4, S6)\}$. Table 2.1 shows the control dependencies that can be determined by examining each of the edges in the set S for the graphs in Figure 2.1.

2.2 Data Dependence Analysis

Data dependencies can be created by scalar variables and elements of arrays. Data dependence analysis consists of global data flow analysis and data dependence testing. The global data flow analysis is used as a framework of data dependence testing to find the relationships between each pair of scalar variables or elements of an array and

Table 2.1: Control dependencies for Figure 2.1.

(a,b) in *S*	Nodes marked	control dependent on	Label
(ENTRY,S1)	*S1, S8*	*ENTRY*	*T*
(S1,S2)	*S2*	*S1*	*T*
(S1,S3)	*S3, S4, S7*	*S1*	*F*
(S4,S5)	*S5*	*S4*	*T*
(S4,S6)	*S6*	*S4*	*F*

to determine the type of potential data dependencies. For two scalar variables, the data dependence testing is very simple. For array elements, subscript analysis must be performed. The remaining part of this section introduces the definitions of data dependencies, global data flow analysis and array element data dependence testing.

2.2.1 Definitions

$IN(S)$ is used to denote the sets of scalar variables and array elements whose values are read by a statement S. $OUT(S)$ is used to denote the set of scalar variables and array elements whose values are modified (or "written") by a statement S. For example, for a statement S :X=Y+Z, $OUT(S) = \{X\}$, and $IN(S) = \{Y,Z\}$. Note that for a loop:

```
S1:  DO I=1,10
S2:       X(I)=A(I+1)*B
S3:  ENDDO
```

$OUT(S2)$={X(1),X(2),...,X(10)}, and $IN(S2)$={A(2),A(3),...,A(11),I,B}. To simplify the notation, we write $OUT(S2)$ = {X(I)}, and $IN(S2)$ = {A(I+1),I,B}.

The three types of data dependencies are defined as follows:

Definition. Given two statements S_i and S_j, S_j is

- *flow-dependent on* S_i, $S_i \delta S_j$, if there is a variable x such that $x \in OUT(S_i) \cap IN(S_j)$ and $x \notin OUT(S_k)$, for $i < k < j$;

- *anti-dependent on* S_i, $S_i\bar{\delta}S_j$, if there is a variable x such that $x \in IN(S_i) \cap OUT(S_j)$ and $x \notin OUT(S_k)$, for $i < k < j$;

- *output-dependent on* S_i, $S_i\delta^o S_j$, if there is a variable x such that $x \in OUT(S_i) \cap OUT(S_j)$ and $x \notin OUT(S_k)$, for $i < k < j$.

To simplify the discussion, we often say that statement S_j is *data-dependent on* S_i, denoted $S_i\delta^* S_j$, if $S_i\delta S_j$ or $S_i\bar{\delta}S_j$ or $S_i\delta^o S_j$. Also, we say that statement S_j is *indirectly data-dependent on* S_i, denoted $S_i\Delta S_j$, if there are statements $S_{k_1}, ..., S_{k_n}, n \geq 0$, such that $S_i\delta^* S_{k_1} \wedge S_{k_1}\delta^* S_{k_2} \wedge ... \wedge S_{k_n}\delta^* S_j$.

When x is a scalar variable, the dependencies due to x can be detected by using data flow analysis, which is discussed later. However, if x is an array element, dependence testing is complicated by the fact that different references to array elements may access the same or different memory locations. It has been shown that the dependence testing problem among array elements is equivalent to the *Integer Linear Programming (ILP) problem* [36], which is an NP-complete problem [44, 16].

To simplify the data dependence testing for array elements, the testing is often limited to loops, and the subscripts of array elements are restricted to linear expressions of the loop index variables. If any one of the subscripts is a nonlinear expression, a dependence is assumed to exist.

Definition [36]. For the following loop:

```
for i1 := L1 to U1 do
  for i2 := L2 to U2 do
    ...
    for in := Ln to Un do
      X(f1(Ī), f2(Ī), ..., fm(Ī)) := .... := X(g1(Ī), g2(Ī), ..., gm(Ī))
    enddo
    ...
```

```
    enddo
  enddo
```

where $\overline{I}$ is the vector $(i_1, i_2, ..., i_n)$, all L_i and $U_i, i = 1, ..., n$, are constants, and $f_j, g_j, j = 1, ..., m$ are known linear functions, two elements of the array X are *dependent* iff there exist index values $i'_1, ..., i'_n$ and $i''_1, ..., i''_n$ such that

$$f_1(\overline{I'}) = g_1(\overline{I''}), ..., f_m(\overline{I'}) = g_m(\overline{I''}),$$

$$L_1 \leq i'_1, i''_1 \leq U_1, ..., L_n \leq i'_n, i''_n \leq U_n.$$

If S is enclosed in n loops with indices $\overline{I} = (i_1, ..., i_n)$, $S^{\overline{I}}$ denotes the instance of S for the iteration $\overline{I}$. Suppose the statements S_i and S_j are enclosed in n loops with indices $\overline{I} = (i_1, ..., i_n)$. Let a vector $\Psi = (\psi_1, \psi_2, ..., \psi_n)$, $\psi_i \in \{<, =, >\}, i = 1, 2, ..., n$ be called a *direction vector*. S_j is *dependent on* S_i *with a direction vector* Ψ, denoted $S_i \delta^*_\Psi S_j$, if there exist iterations $\overline{I'_1} = (i'_1, i'_2, ..., i'_n)$ and $\overline{I''_1} = (i''_1, i''_2, ..., i''_n)$ such that $S_i^{(i'_1, i'_2, ..., i'_n)} \delta^* S_j^{(i''_1, i''_2, ..., i''_n)}$, and the following inequalities hold simultaneously:

$$\begin{array}{l} i'_1 \psi_1 i''_1 \\ i'_2 \psi_2 i''_2 \\ \cdots \\ i'_n \psi_n i''_n \end{array}$$

The vector $(i'_1, i'_2, ..., i'_n) - (i''_1, i''_2, ..., i''_n)$ is called the *direction distance*. Furthermore, S_j is *loop-carried-dependent on* S_i, denoted $S_i \delta^c S_j$, if S_j is dependent on S_i with a direction vector Ψ such that $\Psi = (=, =, ..., =, <, *, *, ..., *)$ and '$*$' denotes '$<$', '$=$' or '$>$'. If $i = j$, we say that S_i is *loop-carried-dependent* on itself.

For example, in the following program:

```
S1:   DO I=1,10
S2:      DO J=1,10
S3:         A(I,J)=B(I,J)
S4:         B(I,J)=A(I,J-1)
S5:         C(I,J)=C(I,J-1)
```

```
S6:       ENDDO
S7:   ENDDO
```

We have $S_3^{(1,1)}\delta S_4^{(1,2)}$ with direction vector $(=,<)$ and direction distance $(0,-1)$ due to array A, and denote it as $S_3\delta_{(=,<)}S_4$, a loop-carried-dependence. S_5 is loop-carried-dependent on itself with direction vector $(=,<)$ and direction distance $(0,-1)$ due to array C. Also, we have $S_3^{(1,1)}\overline{\delta} S_4^{(1,1)}$ with direction vector $(=,=)$ and direction distance $(0,0)$ due to array B.

A loop-carried-dependence means that one statement may store a datum into a location on one iteration of a loop, and another statement may fetch the datum from or store another datum into the location on another iteration of the loop, or vice versa. So, we say a loop is a *carrying dependence loop* if the loop contains a loop-carried-dependence.

2.2.2 Global Data Flow Analysis

Global data flow analysis can be considered as the pre-execution process of ascertaining and collecting information which is distributed throughout a program, generally for the purpose of optimizing the program. It is widely used for code improvements such as analysis of live uses, reaching definitions, available expressions, very busy variables, and data dependencies.

The elimination, or interval, methods and the iterative methods are two popular approaches to global flow analysis [1, 27, 43]. The elimination methods collect the information by continuing to partition the control flow graph of the program into subgraphs, called intervals, and replacing each interval by a single node containing the local information for that interval, until the graph becomes a single node. The iterative

methods propagate the information by initializing the data flow equations to safe values and then iterating the equations until a fixed-point solution is found.

The elimination methods may seem to out-perform the iterative methods, but, when some practical issues, presented in [21], are taken into account, the iterative methods are time competitive with the elimination methods. In addition, the elimination algorithms are usually rather complicated to program. The detailed comparison of the time complexities of these two approaches can be found in [8, 26, 27].

Kildall's algorithm plays an important role in the development of the iterative algorithm because it solves the class of data flow analysis problem in a unified and general lattice theoretic framework [28]. The framework provides a convenient vehicle to analyze the detailed properties of each data flow analysis problem. Hecht and Ullman refined the Kildall's algorithm, and presented a "depth-first" version of Kildall's algorithm [21], a successful iterative algorithm. They introduced a depth-first ordering algorithm for the nodes in a control flow graph, and forced the nodes to be processed in the order. They also proved that their algorithm will finish a global data flow analysis before d+2 iterations, where d is the maximum number of *retreating edges*[1] in a cycle-free path of the control flow graph.

Given a control flow graph G, for each node n of G, $in[n]$ is used to denote its input data stream and $out[n]$ its output data stream. Then, Hecht and Ullman's iterative algorithm performs global data flow analysis in the following two steps:

1. To each node n in G assign an integer number $rPostorder[n]$ and let $out[n] = \{\}$. $rPostorder[n]$ is produced by a depth-first order in which the node n is always visited before its successors except when the node n and its successor form a

[1] A formal definition of *retreating edge* is given in Chapter 4.

retreating edge.

2. Perform the following iteration until no change is made to any node in G where $f_n(x)$ is a data flow function of the node n (the data flow functions f are different for the different applications of data flow analysis, such as Reaching Definition or Live Uses):

```
for each node n in G, in order of rPostorder do
   in[n] := {};
   for each edge (p, n) in G do
      in[n] := in[n] ∪ out[p]
   enddo;
   out[n] := f_n(in[n])
enddo;
```

A more detailed description of the algorithm and the data flow functions $f_n(x)$ is given in [1, 8, 26, 27, 28, 43].

DSA uses an iteration-recursion algorithm, designed for global data flow analysis. This algorithm performs a recursive traversal of the control flow graph of a program in every iteration until it terminates. Hecht and Ullman's depth-first ordering algorithm is also used in this algorithm. These algorithms are described in detail in Chapter 4.

2.2.3 Array Element Dependence Testing

This part overviews algorithms which perform the subscript analysis of the two elements of an array to find the dependence between the two elements. Allen and Kennedy's GCD decision algorithm, Banerjee's inequality decision algorithm, and Burke and Cytron's hierarchical testing algorithm are briefly described in this section. More detailed information can be found in [3, 4, 5, 6, 9, 17, 36, 48].

GCD Decision Algorithm

Let S_1 and S_2 be enclosed in n loops as follows:

for $i_1 := L_1$ **to** U_1 **do**
  for $i_2 := L_2$ **to** U_2 **do**
    ...
    for $i_n := L_n$ **to** U_n **do**
S: $X(..., f(\overline{I}), ...) := ... := X(..., g(\overline{I}), ...)$
    enddo
    ...
  enddo
enddo

where $\overline{I} = (i_1, ..., i_n)$ and

$$f(i_1, ..., i_n) = a_0 + \sum_{k=1}^{n} a_k i_k, \qquad g(i_1, ..., i_n) = b_0 + \sum_{k=1}^{n} b_k i_k.$$

Then, the *low-up bound matrix* LU is defined as:

$$LU = \begin{pmatrix} L_1 & U_1 \\ L_2 & U_2 \\ ... & ... \\ L_n & U_n \end{pmatrix}$$

and the *coefficient matrix* C of the function f and g is defined as:

$$C = \begin{pmatrix} a_0 & b_0 \\ a_1 & b_1 \\ ... & ... \\ a_n & b_n \end{pmatrix}$$

For example, in the following program:

```
S1:  DO I=1,10
S2:      DO J=2,20
S3:          A(20*I+J-20)=B(J,I)
S4:          C(J,I)=A(20*I+J-21)
S5:      ENDDO
S6:  ENDDO
```

the low-up bound matrix is:

$$LU = \begin{pmatrix} 1 & 10 \\ 2 & 20 \end{pmatrix}$$

and the coefficient matrix

$$C = \begin{pmatrix} -20 & -21 \\ 20 & 20 \\ 1 & 1 \end{pmatrix}$$

If S_2 is data-dependent on S_1, then there must exist integers $i'_1, ..., i'_n$ and $i''_1, ..., i''_n$ such that

$$f(i'_1, ..., i'_n) = g(i''_1, ..., i''_n).$$

That is,

$$a_0 + \sum_{k=1}^{n} a_k i'_k = b_0 + \sum_{k=1}^{n} b_k i''_k$$

which can be rewritten

$$\sum_{k=1}^{n} (a_k i'_k - b_k i''_k) = b_0 - a_0.$$

This has an integer solution only when the greatest common divisor of all the left-hand coefficients divide evenly the integer difference on the right-hand side, that is,

$$gcd(a_1, ..., a_n, b_1, ..., b_n) \mid b_0 - a_0.$$

This is the Allen and Kennedy's GCD decision test.

In practice, the GCD test is relatively ineffective, because in most cases the loop index multipliers $a_k = b_k = 1$, so the gcd is 1. However, it is useful in some cases such as

```
S1:  DO I=1,10
S2:      X(2*I)=...=X(2*I+1)
S3:  ENDDO
```

Here $gcd(2, 2) = 2$, and because it is not a divisor of $b_0 - a_0 = 1$, there can be no data dependence.

Banerjee's Inequality Decision Algorithm

Banerjee's inequality decision algorithm depends on the definition of the positive and negative parts of a number as follows:

Definition [3]. Let t be an integer. The *positive part* of the integer t, t^+, and *negative part*, t^-, are defined as:

$$t^+ = \begin{cases} t & \text{if } t \geq 0 \\ 0 & \text{if } t < 0 \end{cases}$$

$$t^- = \begin{cases} -t & \text{if } t \leq 0 \\ 0 & \text{if } t > 0. \end{cases}$$

S_2 is data-dependent on S_1 only if there exist integers $i_1', ..., i_n'$ and $i_1'', ..., i_n''$ such that

$$\sum_{k=1}^{n} (a_k i_k' - b_k i_k'') = b_0 - a_0.$$

If, for a direction vector $\Psi = (\psi_1, \psi_2, ..., \psi_n)$, a lower bound LB and an upper bound UB can be found such that, for each term $k = 1, ..., n$ of the above sum:

$$LB_k^{\psi_k} \leq a_k i_k' - b_k i_k'' \leq UB_k^{\psi_k}$$

where:

if $\psi_k = '*'$ then:

$$LB_k^* = (a_k^- - b_k^+)(U_k - L_k) + (a_k - b_k)L_k$$
$$UB_k^* = (a_k^+ - b_k^-)(U_k - L_k) + (a_k - b_k)L_k$$

if $\psi_k = '<'$ then:

$$LB_k^< = (a_k^- - b_k)^-(U_k - L_k - 1) + (a_k - b_k)L_k - b_k$$
$$UB_k^< = (a_k^+ - b_k)^+(U_k - L_k - 1) + (a_k - b_k)L_k - b_k$$

if $\psi_k = '='$ then:

$$LB_k^{=} = (a_k - b_k)^-(U_k - L_k) + (a_k - b_k)L_k$$
$$UB_k^{=} = (a_k - b_k)^+(U_k - L_k) + (a_k - b_k)L_k$$

if $\psi_k = \ '>'$ then:

$$LB_k^{>} = (a_k - b_k^+)^-(U_k - L_k - 1) + (a_k - b_k)L_k - b_k$$

$$UB_k^{>} = (a_k - b_k^-)^+(U_k - L_k - 1) + (a_k - b_k)L_k - b_k.$$

Summing up these quantities gives the lower and upper bounds, so:

$$\sum_{k=1}^{n} LB_k^{\psi_k} \leq \sum_{k=1}^{n} (a_k i_k' - b_k i_k'') \leq \sum_{k=1}^{n} UB_k^{\psi_k}$$

which can be rewritten as

$$\sum_{k=1}^{n} LB_k^{\psi_k} \leq b_0 - a_0 \leq \sum_{k=1}^{n} UB_k^{\psi_k}$$

If it can be shown that if either $\sum_{k=1}^{n} LB_k^{\psi_k} > b_0 - a_0$ or $\sum_{k=1}^{n} UB_k^{\psi_k} < b_0 - a_0$, then there is no dependence under the constraints of the direction vector $\Psi = (\psi_1, \psi_2, ..., \psi_n)$.

Hierarchical Dependence Testing Algorithm

Burke and Cytron improved the GCD and Banerjee's inequality decision algorithms by introducing hierarchical testing. Hierarchical dependence testing proceeds from a general direction vector ('*') to more specific direction vectors ('<', '=' or '>'). If, at any step, an independence can be shown, the direction vector needs not be refined further. Otherwise, if the direction vector contains any '*' element, '*' is refined to '<', '=' or '>', and the testing continues. If the direction vector does not contain any '*' element, the existence of dependence is assumed under the constraints of the direction vector. Thus, the dependence testing is done on a hierarchy of direction vectors.

Such a hierarchy for two nested loops is shown in Figure 2.2.

Consider the previous example:

```
S1:  DO I=1,10
```

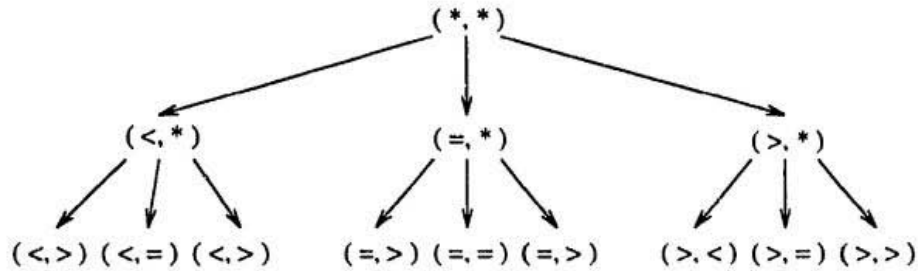


Figure 2.2: Hierarchy of direction vectors for two loops.

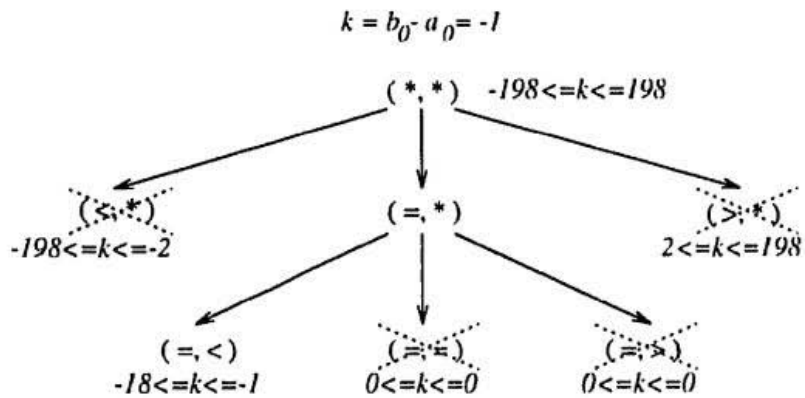


Figure 2.3: The dependence tree for the program.

```
S2:       DO J=2,20
S3:           A(20*I+J-20)=B(J,I)
S4:           C(J,I)=A(20*I+J-21)
S5:       ENDDO
S6:   ENDDO
```

In this example, $L_1 = 1$, $L_2 = 2$, $U_1 = 10$, $U_2 = 20$, $a_0 = -20$, $a_1 = 20$, $a_2 = 1$, $b_0 = -21$, $b_1 = 20$ and $b_2 = 1$. Using Burke and Cytron's hierarchical dependence testing algorithm as the test framework, and Banerjee's inequality decision algorithm to test whether there exists an independence under the constraints of the direction vector, we can obtain the dependence tree shown in Figure 2.3 which indicates the data dependence $S_3 \delta S_4$. Moreover, the direction vector $(=, <)$ indicates that the data dependence $S_3 \delta S_4$ is a loop-carried-dependence.

Another contribution of Burke and Cytron [9] is the linearization of array references, which reduces the complexity of dependence testing. If an array A is declared as

$$A(L_1 : U_1, ..., L_n : U_n)$$

then a reference to an element $A(d_1, ..., d_n)$ can be linearized as $A'(f(d_1, ..., d_n, L_1, ..., L_{n-1}, U_1, ..., U_{n-1}))$, where $f(d_1, ..., d_n, L_1, ..., L_{n-1}, U_1, ..., U_{n-1})$ is the following linear expression:

$$f(d_1, ..., d_n, L_1, ..., L_{n-1}, U_1, ..., U_{n-1}) \ = \ 1 + \sum_{i=1}^{n}((d_i - L_i)\prod_{j=1}^{i-1}(U_j - L_j + 1)).$$

For the following example:

```
S0:  REAL A(20,10),B(20,10),C(20,10)
S1:  DO I=1,10
S2:      DO J=2,20
S3:          A(J,I)=B(J,I)
S4:          C(J,I)=A(J-1,I)
S5:      ENDDO
S6:  ENDDO
```

$A(J, I)$ will be mapped to $A'(20 * I + J - 20)$, and $A(J - 1, I)$ to $A'(20 * I + J - 21)$. The dependence $S_3\delta_{(=,<)}S_4$, obviously, must be preserved.

2.3 Alias Analysis

If two different variables a and b refer to the same memory location, they are called *aliases* of one another. a and b are *explicit aliases* if a programming language construct, such as *union* or *equivalence*, defines them to (partly) overlap. By contrast, they are *implicit aliases* if their aliasing is caused via procedure passing mechanisms. The set of all aliases of x is denoted by $alias(x)$. Note that if y is an alias of x, $y \in alias(x)$ then also x is an alias of y, $x \in alias(y)$.

Explicit aliases can easily be recognized by analyzing the declaration of a program, and hence are not discussed here. The inter-procedural alias analysis is briefly introduced to find implicit aliases. A more detailed presentation is given in [10, 11, 38].

To simplify the discussion, it is assumed that a procedure must begin with a *procedure leader*, which (in FORTRAN) consists of the keyword SUBROUTINE followed by the name of the procedure and a list of formal (or *dummy*) parameters enclosed in parentheses. A procedure is invoked by a CALL statement which consists of the keyword CALL followed by the name of the procedure and a list of arguments enclosed in parentheses. The arguments are also called *actual* parameters. A *global* variable is a nonlocal variable which can be referred to in a procedure body without passing it as a parameter to the procedure. In FORTRAN programs, global variables are those which are declared by COMMON statements.

For example, in the following program[2]:

```
      PROGRAM MAIN
      COMMON G1,G2,G3
S1:   CALL P1(G1,G1,G2)
      END

      SUBROUTINE P1(F1,F2,F3)
      COMMON G1,G2,G3
S2:   CALL P2(F1,F2,F3)
      END

      SUBROUTINE P2(F4,F5,F6)
      COMMON G1,G2,G3
S3:   CALL P1(G3,F4,F5)
S4:   CALL P3(F5,F6)
      END

      SUBROUTINE P3(F7,F8)
```

[2]This is a slightly modified example from [38].

```
COMMON G1,G2,G3
F7=F8+2
END
```

there are three procedures: $P1$ with formal parameters $F1, F2$ and $F3$; $P2$ with formal parameters $F4, F5$ and $F6$; and $P3$ with formal parameters $F7$ and $F8$. These three procedures are invoked by CALL statements S_1, S_2, S_3, and S_4. The actual parameters are $G1, G1$, and $G2$ in S_1, $F1, F2$, and $F3$ in S_2, $G3, F4$, and $F5$ in S_3, and $F5$ and $F6$ in S_4. The global variables in this program are $G1, G2$ and $G3$. This program will be used as an example throughout this section.

Inter-procedural alias analysis finds two types of aliases. *Type-1 aliasing* is caused by using a global variable as an actual parameter in a CALL statement. For example, in S_1, the global variables $G1$ and $G2$ are used as actual parameters of the procedure $P1$. In this case, $G1$ is an alias of the formal parameters $F1$ and $F2$ of $P1$, and $G2$ is $F3$'s alias. So, Type-1 aliasing is also called *global-to-formal aliasing.*

Type-2 aliasing is caused by using the same variable or alias variables more than once as actual parameters in a single CALL statement. For example, in S_1, $G1$ is used two times as the actual parameter in the invocation of $P1$, so the formal parameters $F1$ and $F2$ of $P1$ are aliases of each other. Type-2 aliasing is also called *formal-to-formal aliasing.*

2.3.1 Detecting Type–1 Aliases

The binding graph is the primary data structure to represent the relations between global and formal variables and to calculate Type-1 aliases.

Definition [38]. A *binding graph* is a pair $\beta = (N_\beta, E_\beta)$, where:

1. N_β is the set of formal parameters of all procedures in a program.

2. E_β is a subset of $N_\beta \times N_\beta$ such that an edge (f_1, f_2) is in E_β if there are two procedures p_1 and p_2 such that

   - f_1 is one of the formal parameters of p_1,
   - f_2 is one of the formal parameters of p_2, and
   - f_1 gets bound to f_2 during an invocation of p_2 in p_1.

In the example program, the statement S_2, CALL $P2(F1, F2, F3)$, binds $F1, F2$, and $F3$ to $F4, F5$, and $F6$; the statement S_3, CALL $P1(G3, F4, F5)$, binds $F4$ and $F5$ to $F2$ and $F3$; and the statement S_4, CALL $P3(F5, F6)$, binds $F5$ and $F6$ to $F7$ and $F8$. The binding graph for this example program is shown as Figure 2.4.

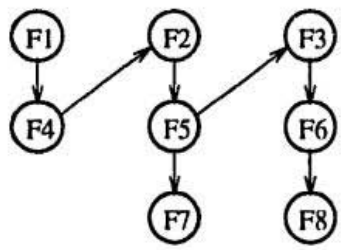


Figure 2.4: The binding graph β for the example program.

Type-1 aliases are determined in the following three steps:

1. Construct the binding graph of the program.

2. For each node f of the binding graph, initialize its alias set, $alias(f)$, as follows:

$$alias(f) := \{g \mid g \text{ is bound to } f\}.$$

3. For each node f_i of the binding graph, propagate the $alias(f)$ sets forward along the directed edges. That is, for each edge (f_i, f_j):

$$alias(f_j) := alias(f_j) \cup alias(f_i).$$

Table 2.2: Type-1 aliases for the example program.

formal parameter f	*alias(f)*
$F1$	$\{G1, G3\}$
$F2$	$\{G1, G3\}$
$F3$	$\{G1, G2, G3\}$
$F4$	$\{G1, G3\}$
$F5$	$\{G1, G3\}$
$F6$	$\{G1, G2, G3\}$
$F7$	$\{G1, G3\}$
$F8$	$\{G1, G2, G3\}$

After step 2, $alias(F1) = \{G1, G3\}$, $alias(F2) = \{G1\}$, $alias(F3) = \{G2\}$ and other *alias* sets are empty. In step 3, these *alias* sets are propagated along the binding graph. The result of Type-1 aliases is shown in Table 2.2.

The binding graph is actually a multi-graph since p_1 may contain several invocations of p_2, which may result in f_1 bound to f_2 more than once. However, for historical reasons, it is still referred to as a graph. Moreover, although standard FORTRAN 77 does not allow recursive calls, many other languages, such as C and PASCAL, do. In this case, it is possible that a binding graph contains cycles, or strongly connected components (SCCs). Therefore, the propagation of aliases in the binding graph should be more complicated than that in the step 3 above. In fact, the main difference between [11] and [38] is how to propagate the *alias* sets among SCCs in step 3. In [11], first, each SCC is reduced to a single node. Then, the *alias* sets are propagated along reduced graph. Finally, the reduced graph is expanded into the original the binding graph. In [38], Tarjan's depth-first search algorithm [46] is used to find SCCs and propagate the *alias* sets along the binding graph at the same time. This improvement simplifies the algorithms. A detailed discussion is given in [11, 38, 46].

2.3.2 Detecting Type–2 Aliases

Type–2 aliases are caused by using alias variables or the same variable more than once as an actual parameter in a single procedure invocation. For example, in $S1$, CALL $P1(G1, G1, G2)$, the variable $G1$ is used twice as an actual parameter for $P1$, so $F1$ and $F2$ are aliases of each other; in $S3$, CALL $P1(G3, F4, F5)$, $G3$ is a Type–1 alias of $F5$, so the formal parameters $F1$ and $F3$ of $P1$ are also aliases.

To detect Type–2 aliases, a set Ω has been proposed [38] (called a *work list*). Each element of Ω is a triple (p, f_1, f_2), indicating that the formal parameters f_1 and f_2 of a procedure p are aliases of each other. When all Type–1 aliases of a program are known, Type–2 aliases can be determined in the following two steps [38]:

1. Construct the initial set Ω of the program.

   For each invocation CALL $p(a_1, ..., a_n)$, and for all actual parameters a_i and a_j, $1 \leq a_i < a_j \leq n$, such that $a_i = a_j$ or a_i is an alias of a_j, the corresponding formal parameters f_i and f_j of the procedure p are added to Ω as a triple (p, f_i, f_j).

2. Expand the set Ω.

   For each triple (p, f_1, f_2) in Ω, check each invocation CALL $q(a_1, ..., a_n)$ in the procedure p, and if there are actual parameters a_i and a_j, $1 \leq i < j \leq n$, such that $f_1 = a_i$ and $f_2 = a_j$, then find the formal parameters f_i' and f_j' of the procedure q and add the triple (p', f_i', f_j') to Ω.

For the example program, step 1 creates the work list Ω as shown in Figure 2.5. The Type–2 aliases of the program, obtained in step 2, are shown in Table 2.3. Combining

Table 2.3: Type-2 aliases for the example program.

formal parameter f	*alias(f)*
$F1$	$\{F2, F3\}$
$F2$	$\{F1, F3\}$
$F3$	$\{F1, F2\}$
$F4$	$\{F5, F6\}$
$F5$	$\{F4, F6\}$
$F6$	$\{F4, F5\}$
$F7$	$\{F8\}$
$F8$	$\{F7\}$

Table 2.2 with Table 2.3 creates the complete aliases for each formal parameter, as shown in Table 2.4.

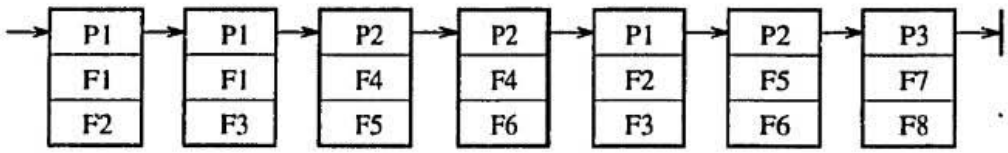


Figure 2.5: The complete work list Ω of the example program.

2.4 Dependence Graph

Both control and data dependencies can be represented as a graph, called a *dependence graph*. The dependence graph G of a program is a directed graph $G = (N, E)$, where the set of nodes, N, is the set of the assignments and branch conditions of the selection and iteration constructs of the program, and the directed edges, $E \subseteq N \times N$, represent both data and control dependencies. An edge (S_i, S_j) is in E if and only if S_j is data-dependent or control-dependent on S_i; if S_j is control-dependent on S_i, (S_i, S_j) is labeled $T(true)$ or $F(false)$, to distinguish them from data-dependent edges. A dependence graph also contains the initial node ENTRY, which, as in control flow graphs, represents a uniform beginning of the execution of the program.

Table 2.4: Complete (Type–1 and Type–2) aliases for the example program.

formal parameter f	*alias(f)*
$F1$	$\{G1, G3, F2, F3\}$
$F2$	$\{G1, G3, F1, F3\}$
$F3$	$\{G1, G2, G3, F1, F2\}$
$F4$	$\{G1, G3, F5, F6\}$
$F5$	$\{G1, G3, F4, F6\}$
$F6$	$\{G1, G2, G3, F4, F5\}$
$F7$	$\{G1, G3, F8\}$
$F8$	$\{G1, G2, G3, F7\}$

For example, in the program shown in Figure 2.6(a), S_2, S_3, S_4 and S_7 are control-dependent on S_1. S_5 and S_6 are control-dependent on S_4. S_5 is data-dependent on S_3 due to P, S_7 is data-dependent on S_6 due to U, and S_8 is data-dependent on S_2 due to Y. The dependence graph G is shown in Figure 2.6(b).

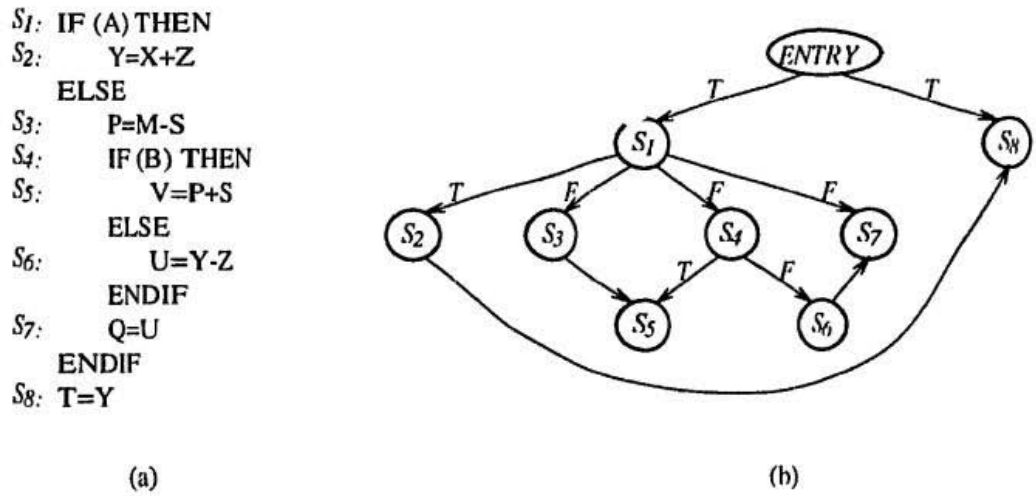


Figure 2.6: An example program and its dependence graph.

Note that this dependence graph is different from the *program dependence graph* as defined in [15]. In a dependence graph, there are no *region nodes* used to summarize the control condition for a node and to group all nodes which have the same set of

control conditions. More information about program dependence graphs can be found in [15, 7].

Chapter 3

Evaluation of Inherent Parallelism

Research in the area of exploiting the parallelism of programs has been conducted over many years, and several experimental compiling systems exploiting parallelism in FORTRAN programs have been developed [2, 3, 7]. However, these systems usually exploit only loop parallelism. Even the recent contributions [32] cannot deal with parallelism between different loops or between a loop and the other parts of a program. This chapter presents a new approach, which can be used to deal with parallelism between loops and between loops and other parts of a program to evaluate the program's inherent parallelism. The presented approach differs quite significantly from the previous work.

The speedup factor is used to evaluate the parallelism of a program. As defined in Chapter 1, the speedup factor of a program is

$$speedup = \frac{T_{serial}}{T_{parallel}},$$

where T_{serial} is the time of the sequential execution of the program, and $T_{parallel}$ is the time of the maximally parallel execution of the same program. Obviously, T_{serial} and $T_{parallel}$ are functions of the program as well as its input data.

Given a program P and its input data D, we can (conceptually) execute the program P with D. In this execution, for each selection construct in P, we can determine the

value of the boolean expression B (the branch condition) which can be TRUE or FALSE. This value is called the *executing value* of the selection construct, and is denoted by b. If the selection construct is nested in n loops with indices $i_1, ..., i_n$, $1 \leq i_j \leq N_j$, $j = 1, ..., n$, there are $N_1 \times ... \times N_n$ executing values for the same boolean expression B; all these values are denoted by $b(i_1, ..., i_n)$. The executing values of all selection constructs of the program P with data D are used for calculating the values T_{serial} and $T_{parallel}$.

3.1 Evaluation of T_{serial}

If a statement S is nested in n $(n \geq 0)$ loops with indices $i_1, ..., i_n$, S's sequential execution time for the iteration $(i_1, ..., i_n)$ is denoted by $T_{serial}(S, (i_1, ..., i_n))$. It is assumed that for an assignment statement S (with no function invocations), the serial execution time does not depend upon the iteration, so $T_{serial}(S, (i_1, ..., i_n)) = T_{serial}(S)$, the serial execution time of the statement S.

The sequential execution times of the sequence, selection and iteration constructs are as follows (t_b is the execution time of the boolean expression B in the selection construct, and $b(i_1, ..., i_n)$ is the executing value of the selection construct):

- *sequence:*

$$T_{serial}(S_1; S_2, (i_1, ..., i_n)) = T_{serial}(S_1, (i_1, ..., i_n)) + T_{serial}(S_2, (i_1, ..., i_n)).$$

- *selection:*

$$T_{serial}(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ endif}, (i_1, ..., i_n)) = \begin{cases} t_b + T_{serial}(S_1, (i_1, ..., i_n)), \\ \qquad \text{if } b(i_1, ..., i_n) = T; \\ t_b + T_{serial}(S_2, (i_1, ..., i_n)), \\ \qquad \text{otherwise.} \end{cases}$$

- *iteration*:

$$T_{serial}(\textbf{for } i := 1 \textbf{ to } n \textbf{ do } S \textbf{ enddo}, (i_1, ..., i_n)) = \sum_{i=1}^{n} T_{serial}(S, (i_1, ..., i_n, i)).$$

An implementation of the evaluation of T_{serial} is described in the next chapter.

3.2 Evaluation of $T_{parallel}$

The evaluation of $T_{parallel}$ is more complicated than that of T_{serial} because it must take into account both control and data dependencies among the statements of a given program. In the proposed approach, the control and data dependencies are represented as a dependence graph, and $T_{parallel}$ is evaluated on the basis of this dependence graph.

For simplicity, in any dependence graph G, an edge (S_i, S_j) is called a *control dependence edge* if S_j is control dependent on S_i. A node S_i is called an IF node if S_i corresponds to a branch condition in a selection construct, and it is called a loop node if S_i corresponds to a branch condition in an iteration construct. Furthermore, a node is called a *control node* if it is an IF node, loop node, or $ENTRY$ node. Note that if an edge (S_i, S_j) is a control dependence edge, S_i must be a control node. A path from $ENTRY$ to S_i is called a *control path* CP_i of S_i if its all edges $(ENTRY, S'_1)$, (S'_1, S'_2), ..., (S'_m, S_i) are control dependence edges.

For each statement S_i of the program, t_i is used to denote the execution time of this statement, and T_i to denote the total execution time of maximally parallel execution of all statements from the beginning of the program to S_i (including S_i). It is assumed that both t_{ENTRY} and T_{ENTRY} are 0.

3.2.1 Evaluation of T_i

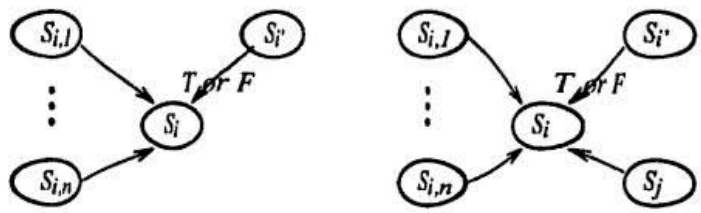
It can be observed that for any given program and any statement S_j, if S_j is data dependent or control dependent on another statement S_i, then S_j must be executed after S_i (if S_j is going to be executed at all). In other words, T_j should be evaluated after the evaluation of T_i. This is the basic principle of the proposed approach.

If a statement S_i is nested in n ($n > 0$) loops with indices $k_1, ..., k_n$, a logical function $F_i(k_1, ..., k_n)$ can be defined in such a way that $F_i(k_1, ..., k_n)$ is TRUE if and only if the statement S_i is executed in the iteration $(k_1, ..., k_n)$, i.e., the control path CP_i of S_i in G is TRUE; otherwise $F_i(k_1, ..., k_n)$ is FALSE.

There are two cases to be considered in evaluating T_i.

Case One: S_i is not included in any loop.

In general case, each node S_i in the dependence graph has n ($n \geq 0$) data dependence edges $(S_{i,j}, S_i), j = 1, ..., n$, and one control dependence edge $(S_{i'}, S_i)$, as shown in Figure 3.1(a).



(a) (b)

Figure 3.1: The dependence graph for node S_i.

In this case:

$$T_i = \begin{cases} t_i + \max(T_{i,1}, ..., T_{i,n}, T_{i'}), & \text{if } F_i; \\ T_{i'}, & \text{otherwise.} \end{cases} \quad (3.1)$$

For example, the program shown in the Figure 3.2(a) can be evaluated using the formula (3.1). Let the executing value b of B in S_2 be TRUE. Then:

```
S1:   P=M-S
S2:   IF(B) THEN
S3:      V=P+S
      ELSE
S4:      U=Y-Z
      ENDIF
S5:   Q=U
```

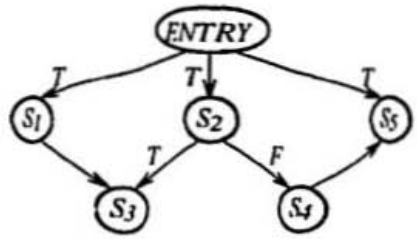


Figure 3.2: The example program 1 and its dependence graph.

$T_1 = t_1$,
$T_2 = t_2$,
$T_3 = t_3 + \max(T_1, T_2) = t_3 + \max(t_1, t_2)$,
$T_4 = T_2 = t_2$, and
$T_5 = t_5 + \max(T_4) = t_5 + t_2$.

On the other hand, if the executing value b of B in S_2 is FALSE, then:

$T_1 = t_1$,
$T_2 = t_2$,
$T_3 = T_2 = t_2$,
$T_4 = t_4 + \max(T_2) = t_4 + t_2$, and
$T_5 = t_5 + \max(T_4) = t_5 + t_4 + t_2$.

Case Two: S_i is in the loop body.

For simplicity, the loop-carried-dependence is limited to a single loop with a normalized index which varies from 1 to N with a unit increment between iterations. Let S_i be loop-carried-dependent on only one S_j in this loop (a similar approach can be extended to more complex loops and more nodes S_j on which S_i is loop-carried-dependent; an implementation of the extension is described in the next chapter). A simple example is as follows:

```
S0:    DO I=1,N
S1:       A(I)=B(I)
```

```
S2:        B(I)=A(I+1)*C(I)
S3:     ENDDO
```

$S1$ is loop-carried-dependent on $S2$, $S2^{1}\delta S1^{2}$, with distance (-1) due to A(I) and A(I+1), and $S2$ is also anti-dependent on $S1$ because of B(I). This program can be unfolded as:

```
A(1)=B(1)
B(1)=A(2)*C(1)
A(2)=B(2)
B(2)=A(3)*C(2)
A(3)=B(3)
B(3)=A(4)*C(3)
...
A(N)=B(N)
B(N)=A(N+1)*C(N)
```

and then it can be observed that, due to the anti-dependencies in A(2), A(3), ... A(N), the statements cannot be executed in parallel.

In general case, each node S_i has n ($n \geq 0$) dependence edges $(S_{i,j}, S_i)$, $j = 1, ..., n$, one control dependence edge $(S_{i'}, S_i)$, and one loop-carried-dependence edge (S_j, S_i) with the direction distance $D = (d)$, as shown in Figure 3.1(b). To deal with the loop-carried-dependence among the iterations of the loop, T_i is replaced by $T_i(k)$ where k is the loop iteration index, $k = 1, ..., n$. Then:

$$T_i(k) = \begin{cases} t_i + \max(T_{i,1}(k), ..., T_{i,n}(k), T_{i'}(k), T_j(k - |d|)), & \text{if } F_i(k) \text{ and } k > |d|; \\ t_i + \max(T_{i,1}(k), ..., T_{i,n}(k), T_{i'}(k)), & \text{if } F_i(k) \text{ and } k \leq |d|; \\ T_{i'}(k), & \text{otherwise.} \end{cases} \tag{3.2}$$

For the above program, $F_i(k), i = 1, 2$, is always TRUE because there is no selection construct in this program. Then:

$$T_1(1) = t_1,$$
$$T_2(1) = t_2 + \max(T_1(1)) = t_1 + t_2,$$

$T_1(2) = t_1 + \max(T_2(1)) = 2t_1 + t_2,$
$T_2(2) = t_2 + \max(T_1(2)) = 2(t_1 + t_2),$
$T_1(3) = t_1 + \max(T_2(2)) = 3t_1 + 2t_2,$
$T_2(3) = t_2 + \max(T_1(3)) = 3(t_1 + t_2),$
...
$T_1(N) = t_1 + \max(T_2(N-1)) = Nt_1 + (N-1)t_2$, and
$T_2(N) = t_2 + \max(T_1(N)) = N(t_1 + t_2).$

That is, the loop cannot be executed in parallel.

Another example for this case is as follows:

```
S0:    DO I=1,N
S1:       A(I)=B(I)
S2:       B(I)=A(I-1)*C(I)
S3:    ENDDO
```

S_2 is loop-carried-dependent on S_1, $S_1^1 \delta S_2^2$, with distance (-1) due to A(I) and A(I-1), and S_2 is also anti-dependent on S_1 because of B(I). Then:

$T_1(1) = t_1,$
$T_2(1) = t_2 + \max(T_1(1)) = t_1 + t_2,$
$T_1(2) = t_1,$
$T_2(2) = t_2 + \max(T_1(2), T_1(1)) = t_2 + \max(t_1, t_1) = t_1 + t_2$
$T_1(3) = t_1,$
$T_2(3) = t_2 + \max(T_1(3), T_1(2)) = t_2 + \max(t_1, t_1) = t_1 + t_2,$
...
$T_1(N) = t_1$, and
$T_2(N) = t_2 + \max(T_1(N), T_1(N-1)) = t_2 + \max(t_1, t_1) = t_1 + t_2.$

Although $S1$ and $S2$ cannot be executed in parallel, the loop contains some parallelism (which can be seen after unfolding the loop). This example shows that the proposed approach detects parallelism existing in loops with loop-carried-dependencies.

The correctness of the proposed approach can be verified by the example shown in Figure 3.3 and used in [32] to illustrate the parallelism available in loops. [32] shows that the loop can be executed in $7N$ time units assuming that an addition is executed

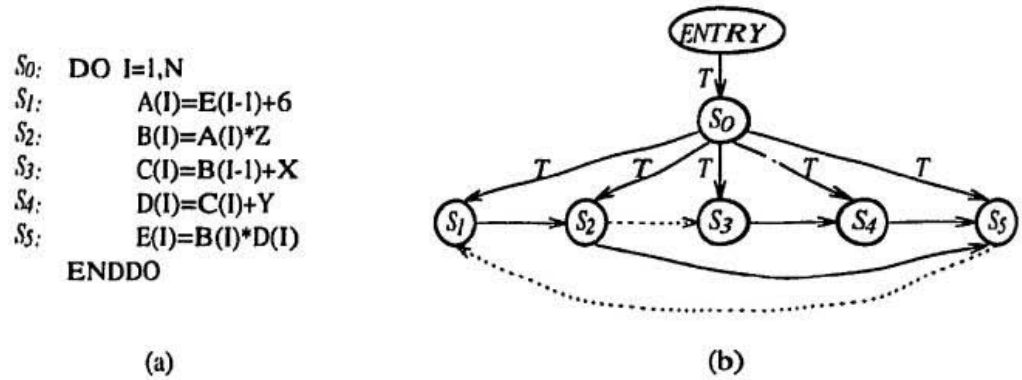


Figure 3.3: The example program 2 and its dependence graph.

in one time unit and a multiplication requires three time units. There are two loop-carried-data-dependencies in this example, that is, $S_5\delta^c S_1$ with the distance (-1) and $S_2\delta^c S_3$ with the distance (-1). Let $t_0 = 0$, $t_1 = t_3 = t_4 = 1$, and $t_2 = t_5 = 3$. The values of $F_i(k), i = 1,5$ are TRUE as there is no selection construct in the program.

For $N = 1$:

$$
\begin{aligned}
&T_1(1) = t_1 = 1,\\
&T_2(1) = t_2 + \max(T_1(1)) = 3 + 1 = 4,\\
&T_3(1) = t_3 = 1,\\
&T_4(1) = t_4 + \max(T_3(1)) = 1 + 1 = 2, \text{ and}\\
&T_5(1) = t_5 + \max(T_2(1), T_4(1))\\
&\qquad = 3 + \max(4,2) = 7 = 1 * 7.
\end{aligned}
$$

For $N = 2$:

$$
\begin{aligned}
&T_1(2) = t_1 + \max(T_5(1)) = 1 + 7 = 8,\\
&T_2(2) = t_2 + \max(T_1(2)) = 3 + 8 = 11,\\
&T_3(2) = t_3 + \max(T_2(1)) = 1 + 4 = 5,\\
&T_4(2) = t_4 + \max(T_3(2)) = 1 + 5 = 6, \text{ and}\\
&T_5(2) = t_5 + \max(T_2(2), T_4(2))\\
&\qquad = 3 + \max(11,6) = 14 = 2 * 7.
\end{aligned}
$$

In general case, using induction on $N(N > 1)$:

$$T_1(N) = 7N - 6,$$

$T_2(N) = 7N - 3$,
$T_3(N) = 7N - 9$,
$T_4(N) = 7N - 8$, and
$T_5(N) = 7N$.

So $T_5(N) = 7 * N$.

3.2.2 Evaluation of $T_{parallel}$

If the analyzed program consists of M statements, then after finding all T_i, $i = 1, ..., M$, the total execution time is:

$$T_{parallel} = \max_{1 \le i \le M}(T_i). \tag{3.3}$$

Moreover, if S_i is nested in n loops with loop indices $i_1, ..., i_n$ such that $1 \le i_j \le N_j, j = 1, ..., n$, then T_i is equal to $\max(T_i(i_1, ..., i_n)), 1 \le i_1 \le N_1, ..., 1 \le i_n \le N_n$.

For example, in the program shown in Figure 3.3(a), since for each $T_i, i = 1, ..., 5$, $T_i(N) = \max(T_i(1), ..., T_i(N))$, and:

$$\begin{aligned} T_{parallel} &= \max(T_1, ..., T_5) \\ &= \max(T_1(N), ..., T_5(N)) \\ &= \max(7N - 6, \\ &\qquad 7N - 3, \\ &\qquad 7N - 9, \\ &\qquad 7N - 8, \\ &\qquad 7N) \\ &= 7 * N. \end{aligned}$$

That is, the same result is obtained as in [32], but using a different approach.

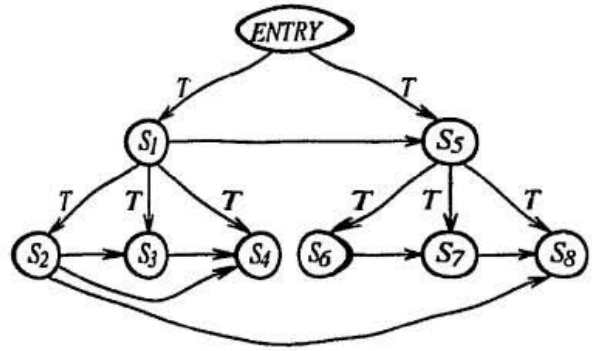
For the program shown in Figure 3.2(a), if B in S_2 is TRUE, then:

$$\begin{aligned} T_{parallel} = \max(T_1, ..., T_5) &= \max(t_1, t_2, t_3 + \max(t_1, t_2), t_2, t_5 + t_2) \\ &= \max(t_3 + \max(t_1, t_2), t_5 + t_2). \end{aligned}$$

On the other hand, if B in S_2 is FALSE, then:

```
S1:  DO I=1,N
S2:      A(I)=E(I)+C(I)
S3:      B(I)=A(I)*D(I)
S4:      C(I)=B(I)+F(I)
     ENDDO
S5:  DO I=1,N
S6:      X(I)=Y(I)*W(I)
S7:      W(I)=X(I)+Y(I)
S8:      Z(I)=A(I)*W(I)
     ENDDO
```

(a)



(b)

Figure 3.4: The example program 3 and its dependence graph.

$$T_{parallel} = \max(T_1, ..., T_5) = \max(t_1, t_2, t_2, t_4 + t_2, t_5 + t_4 + t_2)$$
$$= \max(t_1, t_5 + t_4 + t_2).$$

Note that during the calculation of T_i by using the formulae (3.1) and (3.2), there is no restriction on the positions of statements $S_{i'}$ and $S_{i,j}, j = 1, ..., n$. The statements S_i, $S_{i'}$ and $S_{i,j}, j = 1, ..., n$, can be located in different loops and basic blocks. This means that the proposed approach can be used to evaluate parallelism between loops as well as between a loop and other basic blocks.

For the program shown in the Figure 3.4(a), there is a dependence between the statements in the two loops. Let $t_1 = t_5 = 0$. Since there is no selection construct and no loop-carried-data-dependence in the two loops, then for $i = 1, ..., N$:

$T_1(i) = t_1 = 0,$
$T_2(i) = t_2,$
$T_3(i) = t_2 + t_3,$
$T_4(i) = t_4 + \max(T_2(i), T_3(i)) = t_2 + t_3 + t_4,$
$T_5(i) = t_5 = 0,$
$T_6(i) = t_6,$
$T_7(i) = t_6 + t_7,$ and
$T_8(i) = t_8 + \max(T_2(i), T_7(i)) = \max(t_2, t_6 + t_7) + t_8.$

That is, for each $T_j, j = 1, ..., 8, T_j(1) = ... = T_j(N) = T_j$. So:

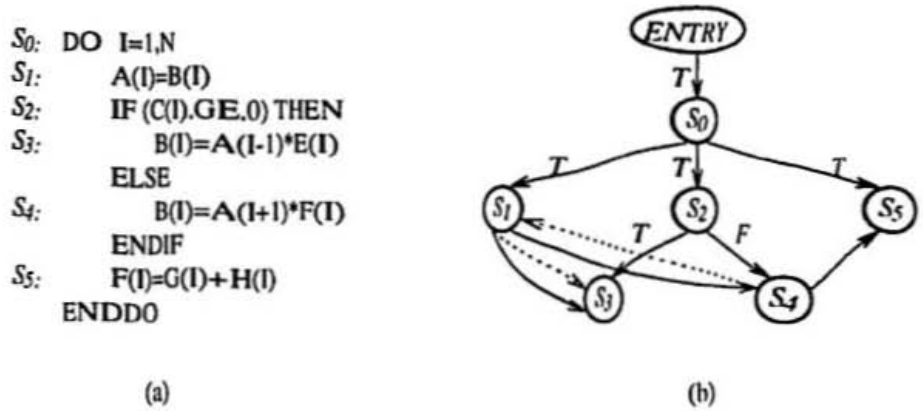


Figure 3.5: The example program 4 and its dependence graph.

$$\begin{aligned} T_{parallel} &= \max(T_1, ..., T_8) \\ &= \max(0, t_2, t_2 + t_3, t_2 + t_3 + t_4, 0, t_6, t_6 + t_7, \max(t_2, t_6 + t_7) + t_8) \\ &= \max(t_2 + t_3 + t_4, \max(t_2, t_6 + t_7) + t_8). \end{aligned}$$

If $t_6 + t_7 \geq t_2$, then $T_{parallel} = \max(t_2 + t_3 + t_4, t_6 + t_7 + t_8)$. That is, the two loops can be executed in parallel, even though there exits the dependence $S_2 \delta S_8$ between the two loops.

The program shown in the Figure 3.5(a) contains a selection construct within an iteration construct. There are two loop-carried-dependencies, $S_1 \delta^c S_3$ with the distance (-1), and $S_4 \delta^c S_1$ with the distance (-1). Suppose that N is equal to 4, and let the executing values of S_2 be $b(1) = T$, $b(2) = F$, $b(3) = F$ and $b(4) = T$. Then:

$$\begin{aligned} &T_1(1) = t_1, \\ &T_2(1) = t_2, \\ &T_3(1) = t_3 + \max(T_1(1), T_2(1)) = t_3 + \max(t_1, t_2), \\ &T_4(1) = T_2(1) = t_2, \\ &T_5(1) = t_5 + \max(T_4(1)) = t_2 + t_5, \\ &T_1(2) = t_1 + \max(T_4(1))) = t_1 + t_2, \\ &T_2(2) = t_2, \\ &T_3(2) = T_2(2) = t_2, \\ &T_4(2) = t_4 + \max(T_1(2), T_2(2)) = t_4 + \max(t_1 + t_2, t_2) = t_1 + t_2 + t_4 \\ &T_5(2) = t_5 + \max(T_4(2)) = t_1 + t_2 + t_4 + t_5, \\ &T_1(3) = t_1 + \max(T_4(2))) = 2t_1 + t_2 + t_4, \end{aligned}$$

$$
\begin{aligned}
&T_2(3) = t_2,\\
&T_3(3) = T_2(3) = t_2,\\
&T_4(3) = t_4 + \max(T_1(3), T_2(3)) = t_4 + \max(2t_1 + t_2 + t_4, t_2) = 2t_1 + t_2 + 2t_4\\
&T_5(3) = t_5 + \max(T_4(3)) = t_5 + 2t_1 + t_2 + 2t_4 = 2t_1 + t_2 + 2t_4 + t_5,\\
&T_1(4) = t_1 + \max(T_4(3)) = t_1 + 2t_1 + t_2 + 2t_4 = 3t_1 + t_2 + 2t_4,\\
&T_2(4) = t_2,\\
&T_3(4) = t_3 + \max(T_1(4), T_2(4), T_1(3))\\
&\qquad = t_3 + \max(3t_1 + t_2 + 2t_4, t_2, 2t_1 + t_2 + t_4) = 3t_1 + t_2 + t_3 + 2t_4,\\
&T_4(4) = T_2(4) = t_2,\\
&T_5(4) = t_5 + \max(T_4(4)) = t_2 + t_5.
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
T_1 &= \max(T_1(1), ..., T_1(4))\\
&= \max(t_1, t_1 + t_2, 2t_1 + t_2 + t_4, 3t_1 + t_2 + 2t_4) = 3t_1 + t_2 + 2t_4,\\
T_2 &= \max(T_2(1), ..., T_2(4)) = \max(t_2, t_2, t_2, t_2) = t_2,\\
T_3 &= \max(T_3(1), ..., T_3(4))\\
&= \max(t_3 + \max(t_1, t_2), t_2, t_2, 3t_1 + t_2 + t_3 + 2t_4) = 3t_1 + t_2 + t_3 + 2t_4,\\
T_4 &= \max(T_4(1), ..., T_4(4)) = \max(t_2, t_1 + t_2 + t_4, 2t_1 + t_2 + 2t_4, t_2) = 2t_1 +\\
&\quad t_2 + 2t_4,\\
T_5 &= \max(T_5(1), ..., T_5(4))\\
&= \max(t_2 + t_5, t_1 + t_2 + t_4 + t_5, 2t_1 + t_2 + 2t_4 + t_5, t_2 + t_5) = 2t_1 + t_2 + 2t_4 + t_5.
\end{aligned}
$$

So:

$$
\begin{aligned}
T_{parallel} &= \max(T_1, ..., T_5)\\
&= \max(3t_1 + t_2 + 2t_4, t_2, 3t_1 + t_2 + t_3 + 2t_4, 2t_1 + t_2 + 2t_4, 2t_1 + t_2 + 2t_4 + t_5)\\
&= \max(3t_1 + t_2 + t_3 + 2t_4, 2t_1 + t_2 + 2t_4 + t_5).
\end{aligned}
$$

3.2.3 Discussion

Program vectorization and parallelization often uses operations called *reduction operations*, such as sum or maximum of a vector or dot products. For example, the sum of an array, say A, is such a reduction operation, as shown in Figure 3.6(a). On a parallel machine, the sum can be calculated in parallel in a binary fashion as shown in Figure 3.6(b), with time complexity $O(t_1 \log_2 n)$. The procedure of this calculation

is equivalent to a transformation of the loop into $O(\log_2 n)$ loops and executing these loops in parallel.

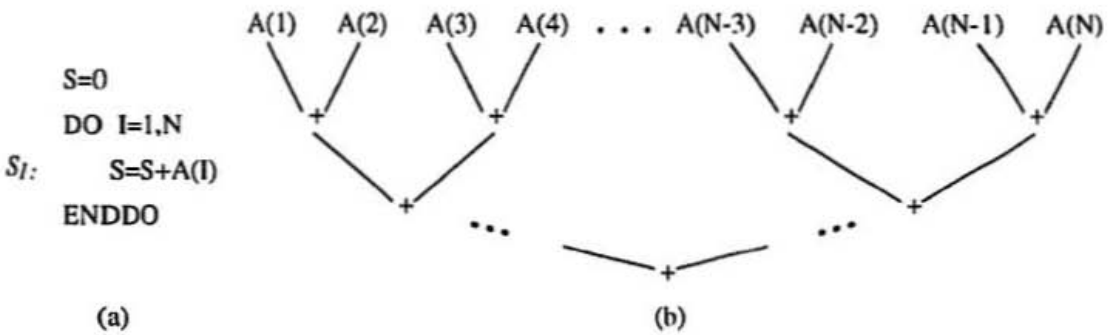


Figure 3.6: The sum of an array and its calculation in a parallel machine.

One of the characteristics of the proposed approach is that the evaluation of the speedup is directly performed on the original program, so there is no need for any transformation of the program. Therefore, the value $O(t_1 \log_2 n)$ cannot be obtained by using the proposed approach. Fortunately, most vector or parallel machines provide instructions to perform reductions [12], and the proposed approach can easily recognize these reduction operations by dependence analysis. Therefore, it is assumed that all reduction operations can be executed on a parallel machine in a constant time, say t, and t can be used to evaluate $T_{parallel}$.

Chapter 4

Implementation of DSA

A program called DSA (Dependence and Speedup Analyzer) was developed and used to perform the dependence analysis and the evaluation of the inherent parallelism of FORTRAN programs. This chapter contains a brief description of DSA and its implementation.

4.1 Overview of DSA

DSA performs the evaluation of the speedup factor of a program in the following three phases: program analysis, dependence analysis and the speedup evaluation.

In the program analysis phase, DSA first reads the source Fortran program and performs its lexical and syntax analysis to collect all the relevant information about the program. Then DSA performs preprocessing for dependence analysis and speedup evaluation. This preprocessing consists of the generation of the control flow graph, calculation of *IN* and *OUT* sets, and alias analysis. The control flow graph is used as an intermediate representation of a program for both dependence testing and speedup evaluation. The *IN* and *OUT* sets are used to detect and determine the types of data dependencies. The alias analysis consists of inter-procedural and intra-procedural alias analyses to expose all implicit aliases created by parameter passing, and explicit aliases

created by equivalence declarations in FORTRAN programs.

After the program analysis, DSA performs control and data dependence analysis and alias analysis based on the information collected during the program analysis. The control dependence testing algorithm proposed by Ferrante [15] has been implemented in DSA. In the data dependence analysis, the iteration-recursion algorithm for global data flow analysis is applied to data dependence testing for both scalar variables and array elements. Three data dependence testing algorithms, Allen and Kennedy's GCD decision algorithm [3], Banerjee and Wolfe's decision algorithm [48], and Burke and Cytron's hierarchical testing algorithm [9], have been implemented in DSA to improve the accuracy of dependence tests. In alias analysis, Cooper and Kennedy's fast inter-procedural alias analysis algorithm based on a binding graph [10, 11, 38] is used. Finally, all control and data dependencies are represented as a dependence graph.

In the speedup evaluation phase, DSA evaluates the inherent parallelism of a program by evaluating its speedup factor. The control flow graph and the dependence graph of the program are used for this evaluation. The evaluation of T_{serial} is performed using the control flow graph, while the evaluation of $T_{parallel}$ is based on the dependence graph. The algorithms presented in Chapter 3 are used to calculate the speedup of programs.

4.2 Program Analysis

Program analysis performs lexical and syntax analysis of the program and preprocessing for dependence analysis and speedup evaluation. During the lexical and syntax analysis, the following information is collected:

- **Variable Table** V: Each element of the Variable Table V is a pair $(id, attr)$, where id is the name of a variable, and $attr$ is a collection of attributes, such as type and the length of the variable.

- **Call Table** C: Each element of the Call Table C is a record which consists of three items: the line number, the name of the caller, and the invocation of the procedure. For the example program of Section 2.3, the Call Table C created by DSA is as follows:

line	*name*	*invocation*
3	MAIN	P1(G1,G1,G2)
8	P1	P2(F1,F2,F3)
13	P2	P1(G3,F4,F5)
14	P2	P3(F5,F6)

- **IF Set** I: Each element of the IF Set I is a triple (c, b, e), where c, b and e are the line numbers of the control, the first and the last lines of each selection construct, respectively.

- **Loop Set** L: Each element of the Loop Set L is a quadruple (c, b, e, v) where c, b and e are the numbers of the control, the first and the last lines of each iteration construct, respectively, and v is a triple (i, f, s) where i and f are the initial and final values of the loop control, and s is the increment.

- **ST Set** T: Each element of the ST Set T is a triple (l, t_s, t_p) where l is the line number of an executable statement S, and t_s and t_p are the execution times for serial and parallel execution of S, respectively.

The preprocessing of dependence analysis includes the generation of the control flow graph, finding the *IN* and *OUT* sets, and alias analysis. The control flow graph is an

intermediate representation of the program and is the basis for the dependence analysis and speedup evaluation. The sets *IN* and *OUT* are used to detect data dependencies and to determine the types of the data dependencies. The alias analysis determines the aliases produced by the procedure passing mechanisms and data equivalences in FORTRAN programs. The result of alias analysis is recorded in the Alias Sets used for data dependence analysis. Since the aliases produced by data equivalences (COMMON and EQUIVALENCE statements) are explicitly declared, these aliases are obtained by directly analyzing the declarations of the programs. The aliases produced by the procedure passing mechanisms are obtained by inter-procedural alias analysis. The algorithms of inter-procedural alias analysis described in Chapter 2 are used in DSA.

4.2.1 Generation of Control Flow Graph

In order to improve the efficiency of DSA, each node of the control flow graph corresponds to a branch condition of a selection or an iteration construct, or to a sequence of consecutive statements called a *block*; the flow of control enters the block only at the beginning and leaves at the end without a possibility of branching except at the end. The generation of the control flow graph of a program consists of the following two steps:

- the nodes of the control flow graph are determined by partitioning the program into blocks, and then
- the set of directed edges of the control flow graph is generated.

Let the first statement of the block be called the *leader* of a block. A source program can be partitioned into blocks by the following two steps:

- the source program is scanned to determine the set of leaders, and
- for each leader, the leader is combined with all following statements up to but not including the next leader or up to the end of the program.

In source FORTRAN programs, the following statements are leaders:

- DO, IF, Assigned GO TO, and Computed GO TO statements,
- statements which immediately follow DO, CONTINUE, IF, ELSE, ENDIF, Assigned GO TO, Computed GO TO, GO TO and RETURN statements,
- statements which have a label except of FORMAT and CONTINUE statements.

Block leaders can easily be identified by scanning the source program. After determining the set of leaders, blocks are obtained by combining each leader with all statements up to but not including the next leader or the end of the program.

In DSA, a node in the control flow graph is a block (not a statement) which is identified by a pair (m, n), where m and n are the line numbers of the first and the last statement of the block, respectively. It is convenient to define two functions, *LNF* and *LNL*, which determine the line number of the first (*LNF*) and the last (*LNL*) statement of each node of the control flow graph. These two functions are used in the next sections of this chapter.

DSA uses a stack for block leaders in order to generate the control flow graph of a FORTRAN program in a single pass. For example, for a FORTRAN DO construct, the line number l_1 of the DO statement is first pushed on the stack. When the first statement of the loop body is processed, its line number l_2 is pushed on the stack, and an edge $< l_1, l_2 >$ of the control flow graph is generated. When the processing of the

loop body is over, and the line number of the last statement of the loop body is l_3, an element l is popped from the stack, and a node n identified by the pair (l, l_3) is generated. Note that if there is no other loop or IF statement in the loop body, then l is l_2; otherwise l is the leader of the last block in the loop body. Finally, when the DOEND is processed with line number l_4, l_1 is popped from the stack and the edges $< l_1, l_4 >$ and $< l_3, l_1 >$ are generated. The other FORTRAN statements are processed in a similar way.

4.2.2 Calculation of *IN* and *OUT* Sets

In a FORTRAN program, the sets *IN* and *OUT* can easily be determined during the syntax analysis of programs if S is an assignment, DO, IF, READ, WRITE, PRINT, or other I/O statement. For instance, if S is an assignment statement $A = expr$, the parser will add A to the set *OUT(S)* and add all variables used in *expr* to the set *IN(S)*. All such updates of the sets *OUT* and *IN* can easily be done during syntax analysis.

However, when S is a CALL statement, or when S contains one or more function invocations, determinating *IN(S)* and *OUT(S)* cannot be done during analysis of S. For example, for a statement CALL $p(X)$, we may know very little about the procedure p during syntax analysis of this statement; X may be used to pass a value to or from p. In this case, there are three possibilities: (1) both *IN(S)* and *OUT(S)* should contain X, (2) only *IN(S)* should contain X, and (3) only *OUT(S)* should contain X. So, when S is a CALL statement or S contains one or more function invocations, the calculation of *IN(S)* and *OUT(S)* must be performed after the syntax analysis is completed. In DSA, all such sets *IN(S)* and *OUT(S)* are determined after the completion of syntax analysis and construction of the binding graph β.

DSA associates two boolean variables *def* and *use* with each formal parameter f of each procedure p. *def* is set TRUE if there is at least one statement S in the procedure p such that $f \in OUT(S)$, otherwise *def* is set FALSE. Similarly, *use* is set TRUE if there is at least one statement S in the procedure p such that $f \in IN(S)$, otherwise it is set FALSE.

Consequently, after the syntax analysis, if S is a CALL statement, or S contains one or more function invocations, the values of *IN(S)* and *OUT(S)* can be determined in two steps: the first step determines the values of *def* and *use* for all formal parameters of all procedures in the program; the second step determines the sets *IN(S)* and *OUT(S)* based on the Call Table and the values of *def* and *use* of all formal parameters.

The following algorithm is used in DSA to determine the values of *def* and *use* for all formal parameters of a program; in this algorithm, for each node n of the control graph, the set *Successors(n)* denotes the set of all nodes connected by directed arcs from n.

Algorithm: Calculating *def* and *use* for all formal parameters.
Input: The Variable Table V, the *IN* and *OUT* sets, and the binding graph $\beta = (N_\beta, E_\beta)$.
Output: The values of *def* and *use* of all formal parameters.

```
for each parameter f in V do
   use(f) := FALSE;
   def(f) := FALSE;
   for each statement S do
      if f ∈ IN(S) then use(f) := TRUE endif;
      if f ∈ OUT(S) then def(f) := TRUE endif
   enddo
enddo;
for each node f in N_β do
   for each h in Successors(f) do
      formalattr(h, tuse, tdef);
      use(f) := use(f) ∨ tuse;
      def(f) := def(f) ∨ tdef
```

```
    enddo
  enddo;

  procedure formalattr(f, tdef, tuse);
  begin
    if Successors(f) is empty then
      tuse := use(f);
      tdef := def(f)
    else
      for each h in Successors(f) do
        formalattr(h, tdef, tuse)
      enddo;
      tuse := use(f) := use(f) ∨ tuse;
      tdef := def(f) := def(f) ∨ tdef
    endif
  end;
```

The *def* and *use* for each of the formal parameters are calculated in the following two steps: (i) the initial values of *def* and *use* are set for each formal parameter f depending on whether or not f is in *IN(S)* or *OUT(S)* of any statement S of the program; step (ii) checks the edges (f_1, f_2) of the binding graph $\beta = (N_\beta, E_\beta)$ and uses a recursive procedure *formalattr* to update the values of $def(f_1)$ and $use(f_1)$.

For the example program from Section 2.3 and its binding graph β shown in Figure 2.4, the initial values of *def* and *use* are: *def(F7)*=TRUE, *use(F8)*=TRUE, and all other vales of *def* and *use* are FALSE. After calculating *use* and *def* for all nodes in the binding graph $\beta = (N_\beta, E_\beta)$, the final values of *def* and *use* are shown in Table 4.1.

For CALL statements, the sets *IN* and *OUT* are determined by the following algorithms:

Algorithm: Finding the sets *OUT* and *IN* for CALL statements.
Input: The Variable Table V, and the Call Table C of a program.
Output: The sets *OUT(S)* and *IN(S)* for each CALL statement S.

Table 4.1: The *def* and *use* values for the example program of Section 2.3.

formal parameter	*def*	*use*
F1	TRUE	TRUE
F2	TRUE	TRUE
F3	FALSE	TRUE
F4	TRUE	TRUE
F5	TRUE	TRUE
F6	FALSE	TRUE
F7	TRUE	FALSE
F8	FALSE	TRUE

for each entry $(n, q, p(a_1, ..., a_n)) \in C$ **do**
  for each argument a_i **in** $(a_1, ..., a_n)$ **do**
    find the ith formal parameter f_i of p in V;
    if *def*(f_i) **then** *OUT(S)* := *OUT(S)* $\cup\{a_i\}$ **endif;**
    if *use*(f_i) **then** *IN(S)* := *IN(S)* $\cup\{a_i\}$ **endif**
  enddo
enddo;

The sets *IN* and *OUT* for the example program of Section 2.3 are shown in Table 4.2.

Table 4.2: The sets *IN* and *OUT* for the example program of Section 2.3.

statement S	*IN(S)*	*OUT(S)*
S_1	$G1, G2$	$G1$
S_2	$F1, F2, F3$	$F1, F2$
S_3	$G3, F4, F5$	$G3, F4$
S_4	$F6$	$F5$

4.3 Dependence Analysis

Dependence analysis performs the analysis of control and data dependencies and represents the dependencies as a dependence graph used in the speedup evaluation. For control dependence analysis, DSA uses the Ferrante [15] control dependence testing algo-

rithm. Burke and Cytron's hierarchical testing algorithm [9] is used as a test framework of Allen and Kennedy's GCD data dependence testing algorithm [3] and Banerjee and Wolfe's inequality data dependence testing algorithm is used for data dependence analysis. Banerjee and Wolfe's algorithm [48] is used to deal with more complicated data dependence testing cases, while the GCD testing is used to improve the efficiency of DSA. Cooper and Kennedy's fast inter-procedural alias analysis algorithm [10, 11, 38] is used for alias analysis. The iteration-recursion algorithm is also implemented for global data flow analysis. Since the algorithms were discussed in detail in Chapter 2, this section discusses only the algorithm for global data flow analysis and provides some details of the Burke and Cytron's hierarchical testing algorithm.

4.3.1 Global Data Flow Analysis

In DSA, the iteration-recursion algorithm has been implemented for global data flow analysis. This section first briefly introduces the Hecht and Ullman's depth-first ordering algorithm as it is used in the iteration-recursion algorithm. The algorithm consists of an initial part and a recursive ordering part. *card(A)* denotes the cardinality of the set A.

Algorithm: Hecht and Ullman's depth-first ordering algorithm.
Input: A control flow graph G with a set of nodes N and an initial node n_0.
Output: A depth-first order *rPostorder*.

---

for each n **in** N **do** *visited[n]* := FALSE **enddo**;
i := card(N);
search(n_0,*i*);

procedure *search*(n, i);
begin
  visited[n] := TRUE;
  for each s **in** *Successors(n)* **do**

```
      if not visited[n] then search(s,i) endif
    enddo;
    rPostorder[n] := i;
    i := i-1
  end;
```

If, for an edge (n, m) in a control flow graph, *rPostorder[n]* ¿ *rPostorder[m]*, the edge is called a *retreating edge* and it indicates the existence of a loop in G. For *rPostorder*, a node is always visited before its successors except when the node and its successor form a retreating edge. Let *indegree[n]* denote the in-degree of node n in the graph G, and let *retreatedge[n]* denote the number of retreating edges directed to n. Moreover, let *fDegree[n] = indegree[n] - retreatedge[n].* The iteration-recursion algorithm uses *fDegree* instead of *rPostorder* to control the order of visited nodes.

The following algorithm determines the data dependence set DD. Each element of DD is a triple (S_i, S_j, x) where x is one of FLOW, ANTI or OUTPUT and indicates that S_j is x-dependent on S_i.

Algorithm: Testing data dependencies.
Input: A control flow graph G with a set of nodes N, an initial node n_0, the sets *fDegree, rPostorder, OUT(S)* and *IN(S).*
Output: The data dependence set DD.

```
f := TRUE;
DD := { };
while f do
    f := FALSE;
    for each n in G do visit[n] := 0 enddo;
    irdf(n0, {}, {}, f,FALSE)
enddo;

procedure irdf(n,tin,tout,flag,retreatedge);
 begin
    dpdttype(n,tin,tout,flag);
    if not retreatedge then
```

```
        visit[n] := visit[n]+1;
        if visit[n]=fDegree[n] then
          for each s in Successors(n) do
            if rPostorder[n] > rPostorder[s] then
              irdf(s,tin,tout,flag,TRUE)
            else
              irdf(s,tin,tout,flag,FALSE)
            endif
          enddo
        endif
      endif
    end;

    procedure dpdttype(n,tin,tout,flag);
    begin
      tmp := DD;
      for i := LNF(n) to LNL(n) do
        for each v in OUT(S_i) do
          for each (v',j) in tout do
            if dp(v,v') then
              DD := DD ∪ {(S_i, S_j, OUTPUT)};
              tout := tout − {(v',j)}
            endif
          enddo;
          for each (v',j) in tin do
            if dp(v,v') then DD := DD ∪ {(S_i, S_j, ANTI)} endif
          enddo;
          tout := tout ∪ {(v,i)}
        enddo;
        for each v in IN(S_i) do
          for each (v',j) in tout do
            if dp(v,v') then
              DD := DD ∪ {(S_i, S_j, FLOW)}
            else
              error("The variable",v,"in line",i,"has no value.")
            endif
          enddo;
          for each (v',j) in tin do
            if dp(v,v') then tin := tin − {(v',j)} endif
          enddo;
```

```
            tin := tin ∪ {(v, i)}
        enddo
    enddo;
    flag := not(tmp = DD)
end;
```

In this algorithm, *LNF(n)* and *LNL(n)* are the functions which determine the line number of the first (*LNF*) and the last (*LNL*) statement associated with the node n (see Section 4.2.1). The formal parameters *tin* and *tout* are used for passing the data flow sets $OUT(S)$ and $IN(S)$ in control flow graph. The iteration control part and the recursive traversal part control the procedure of the global data flow analysis, while the procedure *dpdttype* performs data dependence testing. The boolean function $dp(v, v')$ tests the potential dependence between v and v', checking if v and v' are the same, or if they are aliases of each other, to make S_i and S_j data dependent. If both v and v' are elements of an array, the subscript analysis or array element dependence testing is needed to determine the dependence between them. Array element dependence testing is discussed in the next section.

The iteration-recursion algorithm consists of the iteration control part and the recursive traversal part, shown in the above algorithm which tests data dependencies. For solving other global data flow analysis problems, it is only needed to change the procedure *dpdttype* in the recursive traversal part. The presented algorithm is more efficient than the Hecht and Ullman's iterative algorithm; a formal analysis of the iteration-recursion algorithm can be performed by means of the semi-lattice theory [24, 25, 28, 35], and is given in the Appendix.

4.3.2 Array Element Dependence Testing

The following algorithm is used in DSA for the hierarchical dependence testing. The original algorithm is due to [9].

Algorithm: Hierarchical dependence testing.
Input: The low-up bound matrix LU, the coefficient matrix C, the direction vector V and the position p of the first '*' in V.
Output: A boolean value indicating the existence of dependence.

```
boolean function hierchtest(LU, C, V, p);
begin
  result := FALSE;
  if ddtesting(LU, C, V) then
    if p ≤ card(V) then
      for each c in {'<','=','>'} do
        if not result then
          V[p] := c;
          for i := p+1 to card(V) do V[i] := '*' enddo;
          result := hierchtest(L, U, A, B, V, p+1) ∨ result
        endif;
      hierchtest := result
    else
      hierchtest := TRUE
    endif
  else
    hierchtest := FALSE
  endif
end;
```

hierchtest is a recursive function, invoked by *hierchtest(LU,C,v,1)* with $v = (*, ..., *)$. In the above algorithm, the boolean function *ddtesting* performs Banerjee's inequality decision algorithm as well as Allen and Kennedy's GCD data dependence testing algorithm. Given the low-up bound matrix LU, the coefficient matrix C of two array references and the direction vector v, *ddtesting* performs the GCD and Banerjee's inequality testing. If there is a data dependence, *ddtesting* returns TRUE, otherwise

FALSE is returned. Note that when *hierchtest* returns TRUE, the direction vector v can be analyzed and v in the form (=,...,=,<,*,...,*) indicates a loop-carried-dependence.

4.4 Evaluation of the Speedup Factor

Since the speedup factor of a given program is defined as $\frac{T_{serial}}{T_{parallel}}$, this section describes the algorithms implemented in DSA to evaluate T_{serial} and $T_{parallel}$. The original idea and the detailed discussion of the algorithms are given in Chapter 3.

To evaluate the speedup factor of a given program and to simplify the implementation of DSA, it is assumed that:

1. any simple arithmetic or logical operation is executed in t_s time units,
2. any store or assign memory access operation is executed in t_m time units,
3. any I/O operation takes t_{io} time units,
4. any built-in mathematical FORTRAN function or library function is executed in t_f time units,
5. any statement which contains function invocation can be executed in the time units to execute its operations plus the time units to execute the function.

DSA evaluates T_{serial} and $T_{parallel}$ using these assumptions. It should be pointed out that, in general, it is difficult to estimate the execution times of operations because they depend upon many factors, such as the hardware architecture, data transfer delays and so on. DSA allows the users to modify the values of t_s, t_m, t_{io} and t_f, so users' estimates of execution times can be used.

4.4.1 Evaluation of T_{serial}

The following algorithm is used in DSA to evaluate T_{serial} of a program. The evaluation is performed using the control flow graph G of the program. The ST set T is used to calculate the execution time of each statement. The Loop set L, and IF set I are used to split the program into the sequence, selection and iteration constructs. All these sets are created during the program analysis phase.

Algorithm: Evaluating T_{serial}.
Input: A control flow graph G, the ST Set T, the Loop Set L, the IF Set I.
Output: T_{serial}.

---

function *Tserial(n, i, f, $(i_1, ..., i_v)$)*;
begin
  $l := LNF(n)$;
  if not $(i \leq l \leq f)$ **then**
    $ts := 0$
  else
    if $(l, b, e, (i, f, s)) \in L$ **then**
      $ts := tsb(n)$;
      for $k := i$ **to** f **step** s **do**
        $ts := ts + Tserial(succ(n, \text{TRUE}), b, e, (i_1, ..., i_v, k))$
      enddo;
      $ts := ts + Tserial(succ(n, \text{FALSE}), i, f, (i_1, ..., i_v))$
    else
      if $(l, b, e) \in I$ **then**
        $ts := tsb(n)$;
        if $b(i_1, ..., i_v)$ **then**
          $ts := ts + Tserial(succ(n, \text{TRUE}), b, e, (i_1, ..., i_v))$
        else
          $ts := ts + Tserial(succ(n, \text{FALSE}), b, e, (i_1, ..., i_v))$
        endif;
        $ts := ts + Tserial(succ(succ(n, \text{NIL}), \text{NIL}), i, f)$
      else
        $ts := tsb(n) + Tserial(succ(n, \text{NIL}), i, f, (i_1, ..., i_v))$
      endif
    endif

```
    endif;
    Tserial := ts
end;
```

The function *LNF(n)* returns the line number of the the first statement of the block represented by n (see Section 4.2.1). *succ* is the successor function, so that *succ(n, C)* returns a successor s of the node n with the condition C (which can be TRUE or FALSE). *tsb(n)* is a function used to calculate the T_{serial} time for node n; it just sums all t_s values of the triples (l, t_s, t_p) from the ST set T as long as the line number l is within the basic block n.

This algorithm is defined by a recursive function $Tserial(n, i, f, (i_1, ..., i_v))$, where $(i_1, ..., i_v)$ are the loop indices; *Tserial* calculates the execution time of the block nested in v loops $(i_1, ..., i_v)$, beginning at node n and line i, and extending to line f. The algorithm is invoked by *Tserial(n_0, LNF(n_0), max_line_no,())*, where n_0 is the initial node of the control flow graph. It is assumed that *LNF(ENTRY)* is 0. When the first line associated with node n is out of scope i to f, the algorithm terminates. The rest of the algorithm is composed of three parts for the iteration, selection and the sequence structures. Additional explanations are given in Section 3.1.

4.4.2 Evaluation of $T_{parallel}$

As in Chapter 3, first the algorithm to calculate T_i is presented, and then the evaluation of $T_{parallel}$ is discussed.

Evaluation of T_i

The formula (3.2) is extended to cover more general cases. It is assumed that S_i is nested in loops $L_1, ..., L_p$, data dependent on $S_{i,1}, ..., S_{i,n}$, control dependent on $S_{i'}$, and

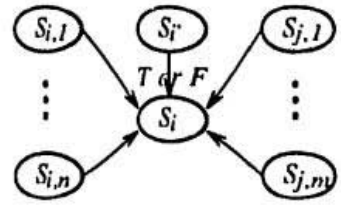


Figure 4.1: The dependence graph for node S_i.

loop-carried-data-dependent on $S_{j,1}$,..., and $S_{j,m}$, as shown in Figure 4.1. For such a case, the formula (3.2) can be extended into the following algorithm. The evaluation is performed using the dependence graph G, which contains all information about the control, data and loop-carried-data-dependencies. The ST set T is used to calculate the execution time of each statement.

Algorithm: Evaluating T_i.
Input: A dependence graph G, the ST Set T; for a node i, i' is the node that i is control-dependent on, and $i_1, ..., i_n$ are the nodes that i is data dependent on, as in Figure 4.1; the execution time of the node i is t_i.
Output: T_i.

---

function $T(i, k_1, ..., k_p)$;
begin
  if i = ENTRY **then**
    $t := 0$
  else
    $t := T(i', k_1, ..., k_p)$;
    if $F_i(k_1, ..., k_p)$ **then**
      for $\ell := 1$ **to** n **do**
        $t := \max(t, T(i_\ell, k_1, ..., k_p))$
      enddo;
      for each S_j such that $S_j \delta^c S_i$ with $D = (d_1, ..., d_p)$ **do**
        if $k_1 > abs(d_1) \wedge ... \wedge k_p > abs(d_p)$ **then**
          $t := \max(t, T(j, k_1 - abs(d_1), ..., k_p - abs(d_p))$
        endif
      enddo;
      $t := t + t_i$
    endif
  endif;

$T := t$
end;

The algorithm follows the formula (3.2). First, it checks the control dependence edge $(S_{i'}, S_i)$ and calculates $T_{i'}$ of the statement $S_{i'}$, as in the *otherwise* case in the formula (3.2). Then, if $F_i(k_1, ..., k_p)$ is TRUE, the max of $T_{i,1}, ..., T_{i,n}$ is calculated by following the data dependence edges $(S_{i,1}, S_i), ..., (S_{i,n}, S_i)$. Finally, if the condition $k_1 > abs(d_1) \wedge ... \wedge k_p > abs(d_p)$ is TRUE, the max of $T_{j,1}, ..., T_{j,m}$ is calculated by following the loop-carried-data-dependence edges $(S_{j,1}, S_i), ..., (S_{j,m}, S_i)$.

Evaluation of $T_{parallel}$

Assuming that a node n in the dependence graph corresponds to a statement S_n, and that S_n is nested in v loops where each loop index satisfies $(i_j, f_j, s_j), j = 1, ..., v$, T_i is equal to $\max(T_i(v_1, ..., v_v))$ where $v_j = 1, ..., (f_j - i_j + 1)/s_j$, $j = 1, ..., v$. Let $m = \prod_{j=1}^{v}(f_j - i_j + 1)/s_j$. An integer k, $1 \leq k \leq m$, can be converted into a v-dimensional vector $(v_1, ..., v_v)$ using the following algorithm:

Algorithm: Mapping.
Input: $k, v, Lidx = \{(i_j, f_j, s_j), j = 1, ...v\}$.
Output: $(v_1, ..., v_v)$.

---

function *mapping(k, v, Lidx)*;
begin
  for $i := v$ **to** 1 **do**
    $t := \prod_{j=1}^{i-1}(f_j - i_j + 1)/s_j$;
    $v_i := (k - 1)/t$;
    if mod$(k, t) = 0$ **then**
      $k := t$
    else
      $k :=$mod(k, t)
    endif
  enddo;
  mapping $:= (v_1, ..., v_v)$

end;

DSA uses the following algorithm to evaluate $T_{parallel}$. For each node n in G, first the values m and v are calculated and each $k, k = 1, ..., m$, is mapped into a v-dimensional vector $idx = (v_1, ..., v_v)$ (using the mapping algorithm above). The vector idx is used in evaluation $T(n, idx)$, that is $T_n(v_1, ..., v_v)$. The maximal value of all $T_n(v_1, ..., v_v)$ is returned as $T_{parallel}$.

Algorithm: Evaluating $T_{parallel}$.
Input: A dependence graph $G = (N, E)$, the Loop Set L.
Output: $T_{parallel}$.

```
function Tparallel;
begin
  t := 0;
  for each node n in N do
    m := 1;
    v := 0;
    Lidx := {};
    for each (c, b, e, (i, f, s)) in L do
      if b ≤ LNF(n) ≤ e then
        m := m * (f − i + 1)/s;
        v := v + 1;
        Lidx := Lidx ∪ {(i, f, s)}
      endif
    enddo;
    for k := 1 to m do
      idx := mapping(k, v, Lidx);
      t := max(t, T(n, idx))
    enddo
  enddo;
  Tparallel := t
end;
```

The function *LNF(n)* returns the line number of the first statement associated with the node n.

Chapter 5

Examples

Five examples are shown in this chapter with their results produced by DSA. Three of these examples are example programs introduced in Chapter 3, where the dependencies and the parallelism in these programs were analyzed in detail. The other two examples are taken from the Livermore Loops [14]. Livermore Loops is a ser of 24 FORTRAN programs selected from real application codes, and run at Lawrence Livermore National Laboratory. These loops have been used extensively to evaluate the performance of computer systems for more than thirty years. Detailed analysis of these loops is presented in [14].

Example 1 illustrates the evaluation of parallelism within a loop which contains IF statements. Example 2 analyzes the parallelism between loops. Example 3 is adopted from [32]; it is used to compare the results obtained from DSA with [32]. The proposed approach is further verified by Example 4 and Example 5, which are taken from the Livermore Loops [14].

In order to simplify the discussion, all operation times t_s, t_m, t_{io} and t_f in Examples 1, 2, 4 and 5 are assumed to be 1 time unit.

5.1 Example 1

The following example program, which corresponds to the example program 4 of Chapter 3 shown in the Figure 3.5(a), illustrates the evaluation of the speedup factor for a simple loop containing an IF statement.

```
 1          REAL A(0:10), B(10), C(10)
 2          REAL G(10), E(10), F(10), H(10)
 3          DO 10 I=1,4
 4              A(I)=B(I)
 5              IF (C(I).GE.0) THEN
 6                  B(I)=A(I-1)*E(I)
 7              ELSE
 8                  B(I)=A(I+1)/F(I)+10
 9              ENDIF
10              F(I)=G(I)+H(I)
11       10 CONTINUE
12          END
```

Let the executing values are as follows:

```
5(1) : T
5(2) : F
5(3) : F
5(4) : T
```

The control flow graph, generated by DSA, is as follows:

```
node (ENTRY): ->3 (T)->STOP (F)
node (3,3): ->4 (T)->STOP (F)
node (4,4): ->5
node (5,5): ->6 (T)->8 (F)
node (6,7): ->10
node (8,9): ->10
node (10,11): ->3
node (12,12): ->STOP
node (STOP)
```

In the above control flow graph, each node is described as `node (ENTRY)`, `node (STOP)`, or `node (n,m)` where `n` and `m` are the line numbers of the program. Each edge is indicated by `->`. The edges with attributes `(T)` or `(F)` are associated with the

selection statements. For example, there are two edges from **node (3,3)**: one to **node (4,4)** with attribute (T), and the other to **node (STOP)**. The ST Set T of this program is as follows:

Line	ts	tp	Statement
3	1	0	DO
4	1	1	ASSIGNMENT
5	2	2	IF
6	3	3	ASSIGNMENT
8	4	4	ASSIGNMENT
10	2	2	ASSIGNMENT

Using the same notation as for the example program 4 in Chapter 3, $t_0 = 0$, $t_1 = 1$, $t_2 = 2$, $t_3 = 3$, $t_4 = 4$ and $t_5 = 2$.

After the dependence analysis, the dependence graph is as follows:

```
node (ENTRY): ->3 (T)
node (3): ->4 (T)->5 (T)->10 (T)
node (4): ->6 (D)->6 (L)(-1)->8 (D)
node (5): ->6 (T)->8 (F)
node (6): ->8 (D)
node (8): ->4 (L)(-1)->10 (D)
node (10):
```

In this dependence graph, each node is described as **node (ENTRY)** or **node (n)** where n is the line number of the program. The edges with attributes (T) and (F) are control-dependence edges with label (T) or (F), the edges with the attribute (D) data-dependence edges, and the edges with the attribute (L)(d) loop-carried-data-dependence edges with distance (d). For example, **node (6)** is control-dependent on **node (5)** with label (T) and **node (8)** is control-dependent on **node (5)** with label (F); **node (4)** is loop-carried-data-dependent on **node (8)** with distance (-1) and **node (10)** is data-dependent on **node (8)**.

The evaluations of T_i, T_{serial}, $T_{parallel}$ and the speedup factor of the program are shown as follows:

```
T(ENTRY) = {0}
T(3) = {0}
T(4) = {1,3,8,13}
T(5) = {2,2,2,2}
T(6) = {5,2,2,16}
T(8) = {2,7,12,2}
T(10) = {4,9,14,4}

Tserial = 39
Tparallel = 16
Speedup Factor = 2.44
```

Here, T(i)=$\{d_1, d_2, \ldots\}$ means that $T_i(1) = d_1, T_i(2) = d_2$, and so on. For comparison, the result derived in Chapter 3 is:

$$T_{parallel} = \max(3t_1 + t_2 + t_3 + 2t_4, 2t_1 + t_2 + 2t_4 + t_5).$$

That is:

$$\begin{aligned} T_{parallel} &= \max(3*1+2+3+2*4, 2*1+2+2*4+2) \\ &= \max(16, 14) = 16. \end{aligned}$$

The evaluation of T_{serial} in Chapter 3 can be used to verify the value of T_{serial}:

$$\begin{aligned} T_{serial} &= 5t_1 + 4(t_2 + t_3 + t_8) + 2t_4 + 2t_6 \\ &= 5*1 + 4(1+2+2) + 2*3 + 2*4 \\ &= 39. \end{aligned}$$

So the speedup factor is indeed equal to 2.44.

5.2 Example 2

This example corresponds to the example program 3 shown in Figure 3.4(a). It illustrates the evaluation of parallelism between loops.

```
1          REAL A(10), B(10), C(10)
2          REAL D(10), E(10), F(10)
3          REAL X(10), Y(10), Z(10), W(10)
4          DO 10 I=1,10
5              A(I)=E(I)+C(I)
```

```
 6              B(I)=A(I)*D(I)
 7              C(I)=B(I)+F(I)
 8        10 CONTINUE
 9           DO 20 I=1,10
10              X(I)=Y(I)*W(I)
11              W(I)=X(I)+Y(I)
12              Z(I)=A(I)*W(I)
13        20 CONTINUE
14           END
```

The ST Set T is as follows:

Line	ts	tp	Statement
4	1	0	DO
5	2	2	ASSIGNMENT
6	2	2	ASSIGNMENT
7	2	2	ASSIGNMENT
9	1	0	DO
10	2	2	ASSIGNMENT
11	2	2	ASSIGNMENT
12	2	2	ASSIGNMENT

Using the same notation as for the example program 3 in Chapter 3, $t_1 = 0$, $t_2 = 2$, $t_3 = 2$, $t_4 = 2$, $t_5 = 0$, $t_6 = 2$, $t_7 = 2$, and $t_8 = 2$. The dependence graph is:

```
node (ENTRY): ->4 (T)->9 (T)
node (4):  ->9 (D)->5 (T)->6 (T)->7 (T)
node (5):  ->6 (D)->7 (D)->12 (D)
node (6):  ->7 (D)
node (7):
node (9):  ->10 (T)->11  (T)->12 (T)
node (10): ->11 (D)
node (11): ->12 (D)
node (12):
```

The values of T_i and the speedup factor of the program as follows:

```
T(ENTRY) = {0}
T(4) = {0}
T(5) = {2,2,2,2,2,2,2,2,2,2}
T(6) = {4,4,4,4,4,4,4,4,4,4}
T(7) = {6,6,6,6,6,6,6,6,6,6}
T(9) = {0}
```

```
T(10) = {2,2,2,2,2,2,2,2,2,2}
T(11) = {4,4,4,4,4,4,4,4,4,4}
T(12) = {6,6,6,6,6,6,6,6,6,6}

Tserial = 142
Tparallel = 6
Speedup Factor = 23.67
```

Comparing with results of Chapter 3:

$$\begin{aligned} T_{parallel} &= \max(t_2 + t_3 + t_4, t_6 + t_7 + t_8) \\ &= \max(2+2+2, 2+2+2) \\ &= 6. \end{aligned}$$

and:

$$\begin{aligned} T_{serial} &= 11t_1 + 10(t_2 + t_3 + t_4) + 11t_2 + 10(t_6 + t_7 + t_8) \\ &= 11 * 1 + 10(2+2+2) + 11 * 1 + 10(2+2+2) \\ &= 142. \end{aligned}$$

So, the speedup factor is indeed 23.67.

5.3 Example 3

This example corresponds to the example program 2 shown in Figure 3.3(a), and is used to compare with the result presented in [32].

```
1           REAL A(10), B(0:10), C(10), D(10), E(0:10)
2           DO 10 I=1,10
3              A(I)=E(I-1)+6
4              B(I)=A(I)*Z
5              C(I)=B(I-1)+X
6              D(I)=C(I)+Y
7              E(I)=B(I)*D(I)
8        10 CONTINUE
9           END
```

Similarly as in Chapter 3 and in [32]:

1. an addition operation is executed in one time unit, and

2. a multiplication operation requires three time units.

Program analysis produces the following ST Set T:

```
 Line     ts    tp   Statement
============================
   2      1     0    DO
   3      1     1    ASSIGNMENT
   4      3     3    ASSIGNMENT
   5      1     1    ASSIGNMENT
   6      1     1    ASSIGNMENT
   7      3     3    ASSIGNMENT
```

The timing data, using the notation from Chapter 3, are: $t_0 = 0$, $t_1 = 1$, $t_2 = 3$, $t_3 = 1$, $t_4 = 1$ and $t_5 = 3$. The dependence graph is:

```
node (ENTRY): ->2 (T)
node (2): ->3 (T)->4 (T)->5 (T)->6 (T)->7 (T)
node (3): ->4 (D)
node (4): ->5 (L)(-1)->7 (D)
node (5): ->6 (D)
node (6): ->7 (D)
node (7): ->3 (L)(-1)
```

T_i and the speedup factor of the program are as follows:

```
T(ENTRY) = {0}
T(2) = {0}
T(3) = {1,8,15,22,29,36,43,50,57,64}
T(4) = {4,11,18,25,32,39,46,53,60,67}
T(5) = {1,5,12,19,26,33,40,47,54,61}
T(6) = {2,6,13,20,27,34,41,48,55,62}
T(7) = {7,14,21,28,35,42,49,56,63,70}

Tserial = 101
Tparallel = 70
Speedup Factor = 1.44
```

In [32], $T_{parallel} = 7 * N = 7 * 10 = 70$, while in Chapter 3, $T_{parallel} = N * (t_5 + t_2 + t_1) = 10 * (3 + 3 + 1) = 70$, so the three results are the same.

5.4 Example 4

The following program is Loop 20 of the Livermore Loops [14]:

```
 1        REAL G(1001), U(1001),V(1001),VX(1001),W(1001)
 2        REAL X(1001),XX(1001),Y(1001),Z(1001)
 3        DO 20 K=1,N
 4            DI=Y(K)-(G(K)/(XX(K)+DK))
 5            DN=0
 6            IF (DI.NE.0) THEN
 7                DN=MAX(S,MIN(Z(K)/DI,T))
 8            ENDIF
 9            X(K)=((W(K)+V(K)*DN)*XX(K)+U(K))/(VX(K)+V(K)*DN)
10            X(K+1)=(X(K)-XX(K))*DN+XX(K)
11       20 CONTINUE
12        END
```

DSA produces the following dependence graph:

```
node (ENTRY): ->3 (T)
node (3): ->4 (T)->5 (T)->6 (T)->9 (T)->10 (T)
node (4): ->6 (D)->7 (D)
node (5): ->7 (D)->9 (D)
node (6): ->7 (T)
node (7): ->9 (D)
node (9): ->10 (D)
node (10): ->9 (L)(-1)
```

If the consecutive executing values for line 6 are T, F, F, T, T, F, F, T, T, F, F, and T, the values of T_{serial}, $T_{parallel}$ and speedup factor, for different values of N, are shown in Table 5.1.

Table 5.1: T_{serial}, $T_{parallel}$ and the speedup factor for Loop 20.

N	4	8	12	16	20	24	28	32	36	40
T_{serial}	97	193	289	385	481	577	673	769	865	961
$T_{parallel}$	64	116	168	220	272	324	376	428	480	532
Speedup Factor	1.52	1.66	1.72	1.75	1.77	1.78	1.79	1.80	1.80	1.81

It can be observed that $T_{parallel} = 12 + 13 * N$ so it is $O(N)$, as in [14].

5.5 Example 5

The following program is Loop 14 of the Livermore Loops [14], a part of a 1-D Particle-in-Cell code.

```
 1          INTEGER N,K,IR(1001),IX(1001)
 2          REAL DEX(1001),DEX1(1001)
 3          REAL EX(1001),EX1(1001),GRD(1001)
 4          REAL RH(1001),RX(1001),VX(1001),XI(1001),XX(1001)
 5          DO 141 K=1,N
 6              VX(K)=0
 7              XX(K)=0
 8              IX(K)=INT(GRD(K))
 9              XI(K)=FLOAT(IX(K))
10              EX1(K)=EX(IX(K))
11              DEX1(K)=DEX(IX(K))
12      141 CONTINUE
13          DO 142 K=1,N
14              VX(K)=VX(K)+EX1(K)+(DEX1(K)*(XX(K)-XI(K)))
15              XX(K)=XX(K)+VX(K)+FLX
16              IR(K)=XX(K)
17              RX(K)=XX(K)-IR(K)
18              IR(K)=MOD2N(IR(K),512)+1
19              XX(K)=RX(K)+IR(K)
20      142 CONTINUE
21          DO 140 K=1,N
22              RH(IR(K))=RH(IR(K))-RX(K)+1.0
23              RH(IR(K)+1)=RH(IR(K)+1)+RX(K)
24      140 CONTINUE
25          END
```

According to [14], the first two DO loops can be combined into a single loop, and the iterations of the loop can be computed in parallel. However, the last DO loop augments elements of RH indexed indirectly through IR. Since the value of IR is unknown at the compile time, the third loop must be executed sequentially. This makes the program's complexity $O(N)$. If it is known that each element of IR is augmented at most once, the third loop could be combined with the first two, and the iterations of the loop could be computed in parallel. So, the complexity would be then $O(1)$.

As indicated in Section 2.2.1, in most of the data dependence testing algorithms, the subscripts of array elements are restricted to linear expressions of the loop index variables, otherwise the existence of dependencies is assumed. Therefore, for this program, DSA assumes that there exists a data dependence from line 22 to line 23 and a loop-carried-data-dependence from line 23 to line 22. The dependence graph is as follows:

```
node (ENTRY): ->5 (T)->13 (T)->21 (T)
node (5): ->13 (D)->6 (T)->7 (T)->8 (T)->9 (T)->10 (T)->11 (T)
node (6): ->14 (D)->15 (D)
node (7): ->14 (D)->15 (D)->16 (D)->17 (D)->19 (D)
node (8): ->9 (D)->10 (D)->11 (D)
node (9): ->14 (D)
node (10): ->14 (D)
node (11): ->14 (D)
node (13): ->21 (D)->14 (T)->15 (T)->16 (T)->17 (T)->18 (T)->19 (T)
node (14): ->15 (D)->19 (D)
node (15): ->16 (D)->17 (D)->19 (D)
node (16): ->17 (D)->18 (D)->19 (D)->22 (D)->23 (D)
node (17): ->18 (D)->19 (D)->22 (D)->23 (D)
node (18): ->19 (D)->22 (D)->23 (D)
node (19):
node (21): ->22 (T)->23 (T)
node (22): ->23 (D)
node (23): ->22 (L)(-1)
```

The values of T_{serial}, $T_{parallel}$ and the speedup factor for different values of N are shown in Table 5.2.

Table 5.2: T_{serial}, $T_{parallel}$ and the speedup factor for Loop 14.

N	4	8	12	16	20	24	28	32	36	40
T_{serial}	151	299	447	595	743	891	1039	1187	1335	1483
$T_{parallel}$	49	77	105	133	161	189	217	245	273	301
Speedup Factor	3.08	3.88	4.26	4.47	4.61	4.71	4.79	4.84	4.89	4.93

Again, it can observed that $T_{parallel} = 21 + 7 * N$, so its complexity is $O(N)$. Since

we cannot assume that each element of IR is augmented at most once, we obtain the same result as that in [14].

Chapter 6

Conclusions

An approach based on dependence analysis is proposed for the evaluation of inherent parallelism of FORTRAN programs. A brief review of the research on dependence analysis and alias analysis is given. A comprehensive description of basic algorithms for dependence analysis is provided which includes Ferrante's control dependence testing algorithm, Allen and Kennedy's GCD, Banerjee and Wolfe's inequality testing and Burke and Cytron's hierarchical data dependence testing algorithm as well as Cooper and Kennedy's fast inter-procedural alias analysis method. Then a concise representation of dependencies, the dependence graph, is introduced. Based on the dependence graph, a general approach to the evaluation of the inherent parallelism of programs is proposed. Finally, an implementation of the control and data dependence testing algorithms, inter-procedural alias analysis algorithms and an approach to evaluation of a program's inherent parallelism is presented.

The main contributions of this work are as follows:

- A general approach to the evaluation of the inherent parallelism available of programs is proposed. The approach can be used to evaluate the parallelism in loops containing selection constructs and dependencies between iterations. Further-

more, the proposed approach can be used to estimate the parallelism between loops or between loops and other parts of a program. The proposed approach can thus deal with parallelism in a very general way.

- A program called DSA was developed for performing dependence analysis and the evaluation of inherent parallelism of programs.

- A global data flow analysis algorithm, the iteration-recursion algorithm, is presented. The algorithm performs a recursive traversal of the control flow graph of a program for a global data flow analysis in every Kildall's iteration. The advantages of this algorithm are that it is easy to implement, and that the control of the order in which nodes are processed and the transfer of information between nodes are more efficient than in other algorithms. For a "structured" program the algorithm can terminate in 2 iterations, which is the best bound for iterative algorithms.

Due to the limited time, there remain some unattended problems and weakness of this project and DSA's implementation.

First, the efficiency of the proposed approach is rather low. The dependencies between iterations are handled by a recursive calculation of loop indices. When the distance of the dependence is equal to 1, this is equivalent to unfolding the loop.

Secondly, DSA cannot analyze the parallelism between procedures as well as functions. Generally speaking, the proposed approach can deal with parallelism between procedures as well as functions, however, DAS assumes that two procedures or functions can be executed in parallel only if there is no dependence between them. Therefore no parallelism between procedures which have dependencies can be analyzed.

Moreover, obtaining the executing values for a given program and its input data may be quite difficult. Although profiling tools and program traces can be used for this purpose, there is no tool which can directly extract the executing values for a given program and its input data.

Finally, some features of the FORTRAN language cannot be analyzed by DSA; they include adjustable arrays, external functions used as actual arguments of other functions, and so on. To address these issues, further research is needed.

Bibliography

[1] Aho, A. U., Sethi, R. and Ullman, J. D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] Allen, F., Burke, M., Charles, P., Cytron, R. and Ferrante, J., "An Overview of the PTRAN Analysis System for Multiprocessing", *Journal of Parallel and Distributed Computing*, 5, 617-640, 1988.

[3] Allen, R. and Kennedy, K., "Automatic Translation of FORTRAN Programs to Vector Form", *ACM Trans. on Programming Languages and Systems*, 4, 491-542, 1987.

[4] Banerjee, U., *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[5] Banerjee, U., "An Introduction to a Formal Theory of Dependence Analysis", *The Journal of Supercomputing*, 2, 133-149, 1988.

[6] Banerjee, U., Eigenmann, R., Nicolau, A., and Padua, D. A., "Automatic Program Parallelization", *Proceedings of the IEEE*, 2, 211-243, 1993.

[7] Baxter, W. and Bauer, H. R., "The Program Dependence Graph and Vectorization", *Proc. 16-th Annual ACM Symp. on Principles of Programming Languages*, 1-11, 1989.

[8] Biswas, B. and Bhattacharjee, G. P., "A Comparison of Some Algorithms for Live Variable Analysis", *International Journal of Computer Mathematics*, 8, 121-134, 1980.

[9] Burke, M. and Cytron, R., "Interprocedural Dependence Analysis and Parallelization", *Proc. SIGPLAN'86 Symp. on Compiler Construction*, 162-175, 1986.

[10] Cooper, K. D. and Kennedy, K., "Interprocedural Side-Effect Analysis in Linear Time", *Proc. SIGPLAN'88 Conf. on Programming Language Design and Implementation*, 57-66, 1988.

[11] Cooper, K. D. and Kennedy, K., "Fast Interprocedural Alias Analysis", *Proc. 16-th Annual ACM Symp. on Principles of Programming Languages*, 49-59, 1989.

[12] Davies, J. R. B., "Issues in Compiler Performance", In: *Performance Evaluation of Supercomputers*, Elsevier Science Publishers B. V. (North-Holland), 51-68, 1988.

[13] Ertel, W., "On the Definition of Speedup", In: *Proc. PARLE'94 Parallel Architectures and Languages Europe* (Lecture Notes in Computer Science 817), Springer Verlag, 289-300, 1994.

[14] Feo, J. T., "An Analysis of the Computational and Parallel Complexity of the Livermore Loops", *Parallel Computing*, 7, 163-185, 1988.

[15] Ferrante, J., Ottenstein, K. J. and Warren, J. D., "The Program Dependence Graph and Its Use in Optimization", *ACM Trans. on Programming Languages and Systems*, 3, 319-349, 1987.

[16] Garey, M. R. and Johnson, D. S., *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, 1979.

[17] Goff, G., Kennedy, K. and Tseng, C., "Practical Dependence Testing", *Proc. ACM SIGPLAN'91 Conf. on Programming Language and Implementation* or *SIGPLAN NOTICES*, 6, 15-29, 1991.

[18] Gupta, R. and Soffa, M. L., "Compilation Techniques for a Reconfigurable LIW Architecture", *The Journal of Supercomputing*, 3, 271-304, 1989.

[19] Gupta, R. and Soffa, M. L., "Region Scheduling: An Approach for Detecting and Redistributing Parallelism", *IEEE Trans. on Software Engineering*, 4, 412-431, 1990.

[20] Hiranandani, S., Kennedy, K. and Tseng, C. W., "Evaluating Computer Optimizations for FOTRAN D", *Journal of Parallel and Distributed Computing*, 21, 27-45, 1994.

[21] Hecht, M. S. and Ullman, J. D., "A Simple Algorithm for Global Data Flow Analysis Problems", *SIAM Journal of Computing*, 4, 519-532, 1975.

[22] Horwitz, S., Demers, A. and Teitelbaum, T., "An Efficient General Iterative Algorithm for Data Flow Analysis", *Acta Informatica*, 24, 679-694, 1987.

[23] Horwitz, S., Prins, J., and Reps, T., "On the Adequacy of Program Dependence Graphs for Representing Programs", *Proc. 15-th Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, 146-157, 1988.

[24] Kam, J. B. and Ullman, J. D., "Global Data Flow Analysis and Iterative Algorithms", *Journal of ACM*, 23, 158-171, 1976.

[25] Kam, J. B. and Ullman, J. D., "Monotone Data Flow Analysis Frameworks", *Acta Informatica*, 7, 305-317, 1977.

[26] Kennedy, K., "A Comparison of Two Algorithms for Global Data Flow Analysis", *SIAM Journal of Computing*, 5, 158-180, 1976.

[27] Kennedy, K., "A Survey of Data Flow Analysis Techniques", In: *Program Flow Analysis: Theory and Applications*, Englewood Cliffs, NJ: Prentice Hall, 5-54, 1981.

[28] Kildall, G., "A Unified Approach to Global Program Optimization", *Proc. ACM Symp. on Principles of Programming Languages*, 194-206, 1973.

[29] Kuck, D. J., Muraoka, and Chen, S. C., "On the Number of Operations Simultaneously Executable in Fortarn-Like Programs and Their Resulting Speed-up", *IEEE Trans. on Computers*, 12, 1293-1310, 1972.

[30] Kuck, D. J., Kuhn, R. H., Leasure, B., Padua, D. A., and Wofe, M., "Dependence Graphs and Compiler Optimizations", *Proc. 8-th Annual ACM Symp. on Principles of Programming Languages*, 207-218, 1981.

[31] Kulkarni, D. and Stumm, M., "Linear Loop Transformatios in Optimizing Compilers for Parallel Machines", *Australian Computer Journal*, 27, 2, 41-50, 1995.

[32] Lilja, D. J., "Exploiting the Parallelism Available in Loops", *IEEE Computer*, 27, 2, 13-26, 1994.

[33] Li, Z., Yew, P. and Zhu, C., 'An Efficient Data Dependence Analysis for Parallelizing Computers", *IEEE Trans. on Parallel Distributed Systems*, 1, 26-34, 1990.

[34] Mahjoub, Z. and Jemni, M., "Restructuring and Parallelizing a Static Conditional Loop", *Parallel Computing*, 21, 2, 339-347, 1995.

[35] Marlowe, T. J. and Ryder, B. G., "Properties of Data Flow Frameworks", *Acta Informatica*, 28, 121-163, 1990.

[36] Maydan, D. E., Hennessy, J. L. and Lam, M. S., "Efficient and Exact Data Dependence Analysis", *Proc. ACM SIGPLAN'91 Conf. on Programming Language and Implementation* or *SIGPLAN NOTICES*, 6, 1-14, 1991.

[37] Maydan, D. E., Hennessy, J. L. and Lam, M. S., "Effectiveness of Data Dependence Analysis", *International Journal of Parallel Programming*, 23, 1, 63-81, 1995.

[38] Mayer, H. G. and Wolfe, M., "IterProcedural Alias Analysis: Implementation and Empirical Results", *Software–Practice and Experience*, 23, 1201-1233, 1993.

[39] Mohd-Saman, M. Y. and Evans, D. J., "Inter-procedural Analysis for Parallel Computing", *Parallel Computing*, 21, 2, 315-338, 1995.

[40] Psarris, K., Klappholz, D. and Kong, X., "On the Accuracy of the Banerjee Test", *Journal of Parallel and Distributed Computing*, 12, 152-157, 1991.

[41] Psarris, K., Kong, X. and Klappholz, D., "The Direction Vector *I* Test", *IEEE Trans. on Parallel Distributed Systems*, 14, 1280-1290, 1993.

[42] Ryder, B. G., "Constructing the Call Graph of a Program", *IEEE Trans. on Software Engineering*, SE-5, 216-226, 1979.

[43] Ryder, B. G. and Paull, M. C., "Elimination Algorithm for Data Flow Analysis", *ACM Computing Surveys*, 18, 277-316, 1986.

[44] Schrijver, A., *Theory of Linear and Integer Programming*, John Wiley & Sons, 1986.

[45] Sun, X-H. and Zhu, J., "Shared Virtual Memory and Generalized Speedup", *Proc. 8-th International Parallel Processing Symposium*, Cancun, Mexico, 637-643, 1994.

[46] Tarjan, R., "Depth-First Search in Linear Graph Algorithms", *SIAM Journal on Computing*, 2, 146-160, 1972.

[47] Tripathi, S. K., "On Detecting Parallelism in Software", *The Journal of Systems and Software*, 1,2, 133-135, 1986.

[48] Wolfe, M., "Automatic Vectorization, Data Dependence, and Optimizations for Parallel Computers", In: *Parallel Processing for Supercomputers and Artificial Intelligence*, 409-440, 1989.

[49] Yang, Y., Ancourt, C. and Irigoin, F., "Minimal Data Dependence Abstractionsns for Loop Transformations: Extended Version", *International Journal of Parallel Programming*, 23, 4, 359-388, 1995.

Appendix A

Iteration–Recursion Algorithms for Global Data Flow Analysis

Three different versions of the iteration-recursion algorithm are presented here. The first version solves the data flow analysis problems as efficiently as the Hecht and Ullman's iterative algorithm. The second version is an improved and specialized version of the first algorithm, which is used for analyzing structured programs. It terminates in no more than 2 iterations, the best bound of Kildall's data flow frameworks. The final algorithm is a combined version of the first and the second versions, which is more efficient than the first version.

A.1 Background

To facilitate discussion of global data flow analysis, the data flow problems are often formulated as instances of a data flow analysis framework, combining flow graph structures with semi-lattice properties [1, 21, 22, 24, 28, 35]. This section introduces the basic concepts and definitions of data flow analysis framework; a more detailed description of these concepts and definitions can be found in [1, 24, 25, 35].

A *directed graph* is usually defined as a pair $G=(N,E)$, where N is a finite set

nodes (the number of nodes in N is denoted $|N|$), and E is a the set of directed edges, $E \subseteq N \times N$. An edge (x,y) in E is *incident from* x and *incident to* y; x is a *predecessor* of y, and y is a *successor* of x. The *indegree* of a node x is the number of predecessors of x, and the *outdegree* of x is the number of successors of x. A *path* from a node n_1 to a node n_k is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ connected by edges (n_i, n_{i+1}) in E for $1 \leq i < k-1$. The *path length* is equal to the number of edges in the path. A path is *simple* if $n_i \neq n_j$ for $i \neq j$. A path is a *cycle* if $n_1 = n_k$ and $k > 1$.

A *flow graph* is a triple $G=(N,E,n_0)$, where (N,E) is a *directed graph*, and n_0 is the *initial node*; there is a path from n_0 to every (other) node in N. The set of all paths from n_0 to a node j is denoted *PATH(j)*.

A *semi-lattice* [35] is a quintuple $L=(A, \underline{0}, \underline{1}, \preceq, \sqcap)$, where:

1. A is a set (often a power set),

2. $\underline{0}$ and $\underline{1}$ are distinguished elements of A,

3. $\preceq$ is a reflexive partial order on A,

4. $\sqcap$ is the meet operation with the following properties:

   - $\sqcap$ is idempotent, commutative and associative,
   - $a \preceq b$ iff $a \sqcap b = a$,
   - $a \prec b$ iff $a \preceq b$ and $a \neq b$,
   - $a \sqcap b \preceq a$,
   - $a \sqcap \underline{0} = \underline{0}$, and
   - $a \sqcap \underline{1} = a$.

A semi-lattice L is *closed*, if it is closed under the operation meet $\sqcap$. A sequence x_1, x_2,..., x_n of elements of L is a *chain* if $x_{i+1} \prec x_i$ for $1 \leq i < n$. L is *bounded* if for each x in L there exists a constant c such that any chain beginning with x has length at most c.

A *data flow analysis framework* [24] is a triple $D=(L, \sqcap, F)$, where L is a bounded semi-lattice with the meet operation $\sqcap$, and F is a family of functions over L such that:

1. Each $f \in F$ distributes over $\sqcap$, i.e., for all x and y in L, $f(x \sqcap y) = f(x) \sqcap f(y)$.

2. There exists an identity function $e \in F$ such that for all $x \in L, e(x) = x$.

3. F is closed under composition, i.e., if $f \in F$ and $g \in F$ then $fg \in F$, where for all $x \in L$, $(fg)(x) = f(g(x))$.

4. For each $x \in L$ there exists a finite subset $H \subseteq F$ such that $x = \sqcap f \in H f(\underline{0})$.

The existence of identity $e \in F$ follows from the fact that a program block can be empty. Closure of F under composition, i.e., for all $f, g \in F$, $fg \in F$, follows from the fact that the concatenation of two blocks is also a program block.

Each function $f \in F$ is *monotone* if

$$(\forall x, y \in L)[x \preceq y \Rightarrow f(x) \preceq f(y)],$$

and a function $f \in F$ is *distributive* if

$$(\forall x, y \in L)[f(x \sqcap y) = f(x) \sqcap f(y)].$$

Every distributive function is monotone.

f^k is used to denote the iterated composition of f, and $f^0 = e$. For each $f \in F$:

$$f^{|k|} = f^0 \sqcap f^1 \sqcap \ldots \sqcap f^{k-1}.$$

f is k *semibounded* for individual functions if for all $x, y \in L$ and for $r > k$:

$$f^r(x) \succeq f^{|k|}(x) \sqcap f^k(y).$$

It has been shown [35] that the k-semiboundness implies that the contribution of the k-th iteration is constant, and many classical intraprocedural problems, such as Reaching Definition, Live Uses, Available Expressions and Very Busy Variables, are 1 semibounded and distributive [21]. A direct result that can be obtained is that it is possible to complete data flow analysis for these classical intraprocedural problems in one iteration. In Kildall's algorithm, every node in a flow graph is visited once per iteration. If the flow graph contains a cycle, the node should be visited at least two times per cycle to complete a round traversal from a node back to the same node. That means that the lower bound of the Kildall's algorithms must be two iterations. It is also shown [35] that if f is 1-semibounded and monotone, then

$$(\forall x, y \in L)[f(y) \succeq y \sqcap x \sqcap f(x)],$$

which is equivalent to

$$(\forall f, g \in F)(\forall x, y \in L)[fg(y) \succeq g(y) \sqcap f(x) \sqcap x],$$

according to Observation 6 in [24]. This property is often used in the proof of data flow analysis algorithms.

An *instance of a data flow analysis framework* [24] $D = (L, \sqcap, F)$ is a pair $I = (G, M)$, where G=(N,E,n_0) is a flow graph, and $M : N \rightarrow F$ is a function which maps each node in N to a function in F.

A.2 Iteration-Recursion Algorithms

The original idea for iteration-recursion algorithms is taken from the data flow algorithm for detection of the data dependencies in a program. The dynamic dependencies between variables can be determined by a recursive algorithm for data flow analysis. However, its time complexity is high and difficult to analyze. To improve the efficiency of this algorithm, the iteration control from traditional iterative algorithms is combined with the recursive traversal.

Three iteration-recursion algorithms are presented in this section. The Hecht and Ullman's "depth-first" version of the Kildall's iterative algorithm [1, 21] is presented first and is compared with the iteration-recursion algorithms.

A.2.1 Hecht and Ullman's Iterative Algorithm

The the Hecht and Ullman's iterative algorithm is as follows.

Algorithm 1: Hecht and Ullman's iterative algorithm.
Input: A particular instance $I = (G, M)$ of data flow analysis framework $D = (L, \sqcap, F)$. where $G = (N, E, n_0)$ is a flow graph with k nodes. Let $N = \{1, 2, ..., k\}$ with nodes ordered by *rPostorder*.
Output: The values $in[n]$ and $out(n)$ for all nodes $n \in N$.

---

for each node n **in** N **do**
  $in[n] := \underline{1}$;
  $out[n] := f_n(\underline{1})$
enddo;
while there are any changes to *out* **do**
  for $n := 1$ **to** k **do**
    $in[n] := \{\}$;
    for each p **in** $Predecessors(n)$ **do**
      $in[n] := in[n] \sqcap out[p]$
    enddo;
    $out[n] := f_n(in[n])$
  end

enddo;

The Hecht and Ullman's improvement to the Kildall's algorithm is in visiting the nodes of a flow graph in the order determined by *rPostorder*.This improvement guarantees that a node is always visited before its successors except when a node and its successor form a retreating edge. It is because of this improvement that the iterative algorithm can terminate in less than $d+2$ iterations where d is the number of retreating edges.

A.2.2 Iteration-Recursion Algorithm One

Let, for each $n \in N$, *indegree*[n] be the in-degree of n, and let *retreatedge*[n] be the number of retreating edges (m, n) incident with n. Let *fDegree*[n] = *indegree*[n] - *retreatedge*[n]. In the following iteration-recursion Algorithm One, the *fDegree* is used to control the order of visited nodes, just like *rPostorder* is used in the Hect and Ullman's algorithm; consequently, the iteration-recursion Algorithm One is as efficient as the Hecht and Ullman's algorithm.

Algorithm 2: Iteration-recursion Algorithm One.
Input: A particular instance $I = (G, M)$ of data flow analysis framework $D = (L, \sqcap, F)$, where $G = (N, E, n_0)$ is a flow graph with k nodes. Let $N = \{1, 2, ...,k\}$ with nodes ordered by *rPostorder*.
Output: The values $in[n]$ and $out(n)$ for all nodes $n \in N$.

---

for each n **in** N **do** $in[n] := \underline{1}$ **enddo;**
f := FALSE;
while not f **do**
  f := TRUE;
  for each node n **in** N **do** $visit[n] := 0$ **enddo;**
  $irdf1(n_0, in[n_0], f,$ FALSE)
enddo;

procedure $irdf1(n, x, flag, retreatedge)$;

```
begin
   tmp := in[n];
   in[n] := in[n] ⊓ x;
   flag := flag and (tmp = in[n]);
   if not retreatedge then
      visit[n] := visit[n]+1;
      if visit[n]=fDegree[n] then
         out[n] := f_n(in[n]);
         for each s in Successors(n) do
            if rPostorder[n] >rPostorder[s] then
               irdf1(s, out[n], flag, TRUE)
            else
               irdf1(s, out[n], flag, FALSE)
            endif
         enddo
      endif
   endif
end;
```

The iteration control part of Algorithm One is the same as in the Hecht and Ullman's algorithm. In the recursive traversal part, the condition *visit[n] = fDegree[n]* is used to guarantee that the nodes are processed before their successors, with the exception of the retreating edges.

Comparing this algorithm with the Hecht and Ullman's iterative algorithm, it is obvious that if the Hecht and Ullman's algorithm terminates in less than $d+2$ iterations, the Algorithm One will also terminate in $d+2$ iterations because both algorithms process the nodes of the flow graph before their successors except for retreating edges. In addition, Algorithm One has two other advantages. One is that passing the information between nodes is more efficient, so the value $in[n]$ can be determined by $in[n] := in[n] \sqcap x$ rather than by visiting the predecessors of n which is the case in the Hecht and Ullman's algorithm. The other advantage is that Algorithm One can control the processing order of nodes more efficiently. The iteration-recursion Algorithm Two is a result of using

this advantage.

A.2.3 Iteration-Recursion Algorithm Two

The iteration-recursion Algorithm Two is more efficient in data flow analysis than Algorithm One, however, the programs to be processed must be "structured", so each loop in the program can have only one exit. For structured programs, the flow graphs can be constructed in such a way that for each loop condition, the left successor is always the loop body while the right successor corresponds to the loop exit. Algorithm Two is an improved and specialized version of Algorithm One which takes advantage of this representation of flow graphs.

Algorithm 3: Iteration–recursion Algorithm Two.
Input: A particular instance $I = (G, M)$ of data flow analysis framework $D = (L, \sqcap, F)$, where $G = (N, E, n_0)$ is a flow graph with k nodes. Let $N = \{1, 2, ..., k\}$ with nodes ordered by *rPostorder*.
Output: The values $in[n]$ and $out(n)$ for all nodes $n \in N$.

```
for each n in N do in[n] := 1 enddo;
for i := 1 to 2 do
   for each node n in N do visit[n] := 0 enddo;
   irdf2(n0, in[n0], FALSE)
enddo;

procedure irdf2(n, x, retreatedge);
begin
   in[n] := in[n] ⊓ x;
   out[n] := fn(in[n]);
   if not retreatedge then
      visit[n] := visit[n]+1;
      if visit[n]=fDegree[n] ) then
         for each s in Successors(n), from left to right do
            if rPostorder[n] >rPostorder[s] then
               irdf2(s, out[n], TRUE)
            else
               irdf2(s, out[n], FALSE)
```

```
        endif
      enddo
    endif
  endif
end;
```

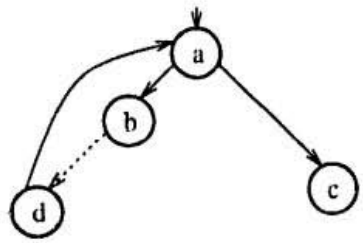


Figure A.1: Flow graph of a loop in a "structured" program.

Figure A.1 shows the flow graph of a loop in a "structured" program. Node a is the condition node of a loop. The left branch is the loop body and the right branch is the exit from the loop. During processing of node a, $out[a]$ is passed to b and b is visited. After processing the loop body, a is visited again along the retreating edge (d, a). At this time, the values $in[a]$ and $out[a]$ are assigned again, and c is selected for processing. It is important that the value $out[a]$ now contains information from the loop body b to d; it is this change of the value $out[a]$ that allow $irdf2$ to make two iterations only.

Formal analysis of Algorithm Two can be performed on the basis of the following theorems. Let $\mathrm{PATH}^n(j) = \{\ p \mid p$ is the path which Algorithm Two follows from node n_0 to node j in the n-th iteration $\}$.

Theorem 1. If $p \in PATH(j)$ and $q \in PATH^2(j)$ and p is a simple path in a flow graph and q contains every node in p and one cycle in the flow graph, then $f_p(\underline{1}) \succeq f_q(\underline{1})$.

In the following proof, let $f_p(x) = f_m(f_{m-1}(...f_1(x)...))$ if $p = (1, ..., m-1, m)$ is a path of the flow graph.

Proof: If $p \in PATH(j)$ and $q \in PATH^2(j)$ such that p is a simple path in a flow

graph, and q contains every node in p and one cycle in the flow graph, then there exists a node i in the path q such that $q = n_0...i...i...j$, and $p = n_0...i...j$. That is, there is a retreating edge which is incident to i. Let $a = n_0...i, b = i...i$, and $c = i...j$. According to the Algorithm Two, there must exist an x, passed to i along the retreating edge, such that $f_c(f_b(f_a(\underline{1}))) = f_c(f_a(\underline{1}) \sqcap x)$. So:

$$\begin{aligned} f_p(\underline{1}) &= f_{cba}(\underline{1}) \\ &= f_c(f_a(\underline{1})) \\ &\succeq f_c(f_a(\underline{1}) \sqcap x) \\ &= f_c(f_b(f_a(\underline{1})) \\ &= f_{cba}(\underline{1}) \\ &= f_q(\underline{1}). \quad \square \end{aligned}$$

Theorem 2. Let $D = (L, \sqcap, F)$ be a data flow analysis framework. When Algorithm Two terminates, then for each node j in N and for each path p in $PATH(j)$, there exist paths $q_1, ..., q_r$ each in $PATH^2(j)$ such that $f_p(\underline{1}) \succeq \sqcap 1 \leq i \leq r f_{q_i}(\underline{1})$ if D satisfies the following condition:

$$(\forall f, g \in F)(\forall x, y \in L)[fg(y) \succeq g(y) \sqcap f(x) \sqcap x]. \tag{A.1}$$

Proof: If the condition (A.1) is satisfied, for each node $j \in N$ and each path $p \in PATH(j)$, there are three cases to consider.

Case 1. p is a simple path in the flow graph. There exists a path $q \in PATH^2(j)$ such that q contains p since Algorithm Two traverses every edge in the flow graph. If q is also a simple path in the flow graph, then $q = p$, and $f_p(\underline{1}) \succeq f_q(\underline{1})$. Otherwise a cycle must exist in q, so *Theorem 1* can be applied to p and q, and then $f_p(\underline{1}) \succeq f_q(\underline{1})$.

Case 2. p contains one cycle. There must exist a node i in p such that $p = n_0...i...i...j$. Let $p_1 = n_0...i$, $p_2 = i...i$, and $p_3 = i...j$. There is a path $q \in PATH^2(j)$ such that $q = n_0...i...i...j$ contains p. Let $q_1 = n_0...i$, $q_2 = i...i$, and $q_3 = i...j$. As in Case 1, $f_{p_1}(\underline{1}) \succeq f_{q_1}(\underline{1})$, $f_{p_2q_1}(\underline{1}) \succeq f_{q_2q_1}(\underline{1})$, and $f_{p_3q_2q_1}(\underline{1}) \succeq f_{q_3q_2q_1}(\underline{1})$, so:

$$\begin{aligned}
f_p(\underline{1}) &= f_{p_3p_2p_1}(\underline{1}) \\
&= f_{p_3}(f_{p_2}(f_{p_1}(\underline{1}))) \\
&\succeq f_{p_3}(f_{p_2}(f_{q_1}(\underline{1}))) \\
&= f_{p_3}(f_{p_2q_1}(\underline{1})) \\
&\succeq f_{p_3}(f_{q_2q_1}(\underline{1})) \\
&= f_{p_3q_2q_1}(\underline{1}) \\
&\succeq f_{q_3q_2q_1}(\underline{1}) \\
&= f_q(\underline{1}).
\end{aligned}$$

Case 3. p contains more than one cycle. Let $p = n_0, ..., i_a, ..., i_b, ..., i_a, ..., j$ such that $i_a = i_b$. Let $p' = n_0, ..., i_a$, $p'' = i_a, ..., i_b$, $p''' = i_b, ..., i_a$, $p'''' = i_a, ..., j$, and $x = y = f_{p'}(\underline{1})$ in (A.1) (p'' and p''' may be different), then p can be decomposed into three paths p_1, p_2 and p_3 which contain a smaller number of cycles than p does.

$$\begin{aligned}
fp(\underline{1}) &= f_{p''''p'''p''p'}(\underline{1}) \\
&= f_{p''''}(f_{p'''}(f_{p''}(f_{p'}(\underline{1})))) \\
&\succeq f_{p''''}(f_{p'''}(f_{p'}(\underline{1})) \sqcap f_{p''}(f_{p'}(\underline{1})) \sqcap f_{p'}(\underline{1})) && \textit{by assumption} \\
&= f_{p''''}(f_{p'''}(f_{p'}(\underline{1}))) \sqcap f_{p''''}(f_{p''}(f_{p'}(\underline{1}))) \sqcap f_{p''''}(f_{p'}(\underline{1})) && \textit{by distribution} \\
&= f_{p''''p'''p'}(\underline{1}) \sqcap f_{p''''p''p'}(\underline{1}) \sqcap f_{p''''p'}(\underline{1}) \\
&= f_{p_1}(\underline{1}) \sqcap f_{p_2}(\underline{1}) \sqcap f_{p_3}(\underline{1}).
\end{aligned}$$

If p_1, p_2 and p_3 contain only one or zero cycles, then there are q_1, q_2 and $q_3 \in PATH^2(j)$ such that $f_{p_1}(\underline{1}) \succeq f_{q_1}(\underline{1})$, $f_{p_2}(\underline{1}) \succeq f_{q_2}(\underline{1})$, and $f_{p_3}(\underline{1}) \succeq f_{q_3}(\underline{1})$, as in Cases 2 and 1. So, $f_p(\underline{1}) \succeq f_{q_1}(\underline{1}) \sqcap f_{q_2}(\underline{1}) \sqcap f_{q_3}(\underline{1})$. Otherwise the decomposition is continued until every $p_i (i = 1, ..., r)$ contains at most one cycle, and then $f_p(\underline{1}) \succeq \sqcap 1 \leq i \leq r f_{q_i}(\underline{1})$ for $q_i (i = 1, ..., r)$. □

According to *Theorem 2*, if the condition (A.1) is satisfied, Algorithm Two terminates in two iterations with the correct results of the data flow analysis. As indicated earlier,many intraprocedural data flow problems satisfy condition (A.1).

A.2.4 Iteration-Recursion Algorithm Three

The main difference between Algorithms One and Two is in the way in which the value of $out[n]$ is calculated. This difference is the major reason that Algorithm Two is more efficient than Algorithm One. The following algorithm is a combination of Algorithms One and Two.

Algorithm 4: Iteration-recursion Algorithm Three.
Input:. A particular instance $I = (G, M)$ of data flow analysis framework $D = (L, \sqcap, F)$, where $G = (N, E, n_0)$ is a flow graph with k nodes. Let $N = \{1, 2, ...,k\}$ with nodes ordered by *rPostorder.*
Output: The values $in[n]$ and $out(n)$ for all $n \in N$.

---

for each node n **in** N **do** $in[n] := \underline{1}$ **enddo**;
f := FALSE;
while not f **do**
  f := TRUE;
  for each node n **in** N **do** $visit[n] := 0$ **enddo**;
  $irdf3(n_0, in[n_0], f,$ FALSE)
enddo;

procedure $irdf3(n, x, flag, retreatedge)$;
begin
  $tmp := in[n]$;

```
    in[n] := in[n] ⊓ x;
    out[n] := f_n(in[n]);
    flag := flag and (tmp = in[n]);
    if not retreatedge then
      visit[n] := visit[n]+1;
      if visit[n]=fDegree[n] then
        for each s in Successors(n) do
          if rPostorder[n] >rPostorder[s] then
            irdf3(s, out[n], flag, TRUE)
          else
            irdf3(s, out[n], flag, FALSE)
          endif
        enddo
      endif
    endif
  end;
```

In the best case, when all nodes are ordered "properly", the algorithm terminates in not more than 3 iterations. The worst case is when the algorithm terminates in $d+2$ iterations with the $(d+2)*r$ visits to the nodes, where r is the number of the retreating edges in the flow graph. Normally, the number of the nodes of a flow graph is much greater than the number of nodes incident with retreating edges. In such cases, the time spent on the $(d+2)*r$ visits is much smaller than that spent on other computations. It can thus be assumed that Algorithm Three is more efficient than Algorithm One.

A.3 Conclusions

Algorithm One performs data flow analysis as efficiently as the Hecht and Ullman's "depth-first" version of the Kildall's algorithm. Algorithm Two completes the intraprocedural analysis in two iterations, which is the lower bound of the Kildall's algorithm for "structured" programs. Algorithm Three is more efficient than Algorithm One. All three algorithms are easy to implement.