DATA DEPENDENCE ANALYSIS FOR FORTRAN PROGRAMS IN THE *IMPACT* COMPILER

BY

GRANT EDWARD HAAB

B.S., University of Illinois at Urbana-Champaign, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. I consider myself very fortunate to have the opportunity to work with someone of his caliber.

This project would not have been possible without the support of the entire *IMPACT* research group. Nancy Warter, Krishna Subramanian, Ben-Chung Cheng and David August helped me to develop the *Pcode* module, which is based on the *Hcode* module developed by Pohua Chang. Krishna Subramanian, Sadun Anik, Yoji Yamada, Dan Lavery, Teresa Johnson and Professor Chung-Ta King developed optimizations and code transformations using the *Pcode* dependence analysis package. David Gallagher built a *C*-language dependence analysis package using much of my existing implementation and propagated the data dependence information through the *Hcode* and *Lcode* modules. Thanks to everyone in the *IMPACT* research group who contributed to this work through discussions, comments, and implementation.

Two other research groups figured prominently in the development of the *IMPACT* data dependence analyzer. Many thanks to Professor William Pugh and his research group for developing the Omega data dependence test and for their many comments,

suggestions, and bug fixes. In addition, thanks to Bob Rau, Mike Shlansker and Vinod Kathail of Hewlett-Packard Laboratories for their valuable insights regarding data dependence analysis and code optimization.

Thanks to the Fannie and John Hertz Foundation, which awarded me a Graduate Fellowship, providing generous financial support for my graduate studies.

Finally, I would like to thank my spouse Matthew, my brothers Greg, Galen, and Gavin, Matt's sister Jennifer and brother in-law Michael, my parents, Paul and Kathy, and Matt's parents Paul and Joann, for their love and support during my graduate studies.

DEDICATION

To my spouse Matthew Hesson-McInnis for your love, friendship and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Optimizing compilers have progressed to the point that they require detailed program information to perform aggressive optimization. Program transformations prove invaluable for memory system optimization and program parallelization. To transform programs without invalidating their results, an optimizing compiler must establish the data dependence relationships among the variables accesses in the program. Data dependence relationships summarize the flow of values through the program and provide constraints on program restructuring.

For *Fortran* programs, the transformations rely heavily on the dependence relationships of array variables in the presence of nested loops. Although a standard data flow analysis can determine some of these relationships for scalar variables, more sophisticated techniques must be utilized for array variables to take into account the array subscript information and loop bound and nesting information.

This thesis presents the design and implementation of an effective data dependence analysis package for *Fortran* programs within the framework of the *IMPACT* compiler [1].

The data dependence analysis package is built around the Omega Test [2], which solves linear diophantine equations to determine the program data dependence relationships. However, extensive program analysis and data structure preparation are necessary prerequisites for a data dependence analyzer to provide useful information for loop transformations. Subscript-by-subscript data dependence analysis for arrays, handling of array subscript variables that are not loop indices, and variable aliasing considerations are a few of the features which must be present to effectively analyze real programs. Furthermore, the program references must be organized, classified, and analyzed before data dependence analysis is applied. Finally, a data dependence analyzer must be able to store the analysis results in a reasonably compact and effective representation. The *IMPACT* data dependence analysis package satisfies all these requirements and provides compiler writers with an effective analysis framework for enabling valid program transformations for optimization.

The rest of this thesis is organized into several chapters. Chapter 2 presents the formal background material for a clearer understanding of data dependence analysis. An overview of the *IMPACT* compiler focusing on the modules relevant to data dependence analysis is given in Chapter 3. Chapter 4 presents the variable reference analysis process, which is necessary to support data dependence analysis as well as program transformations. The array data dependence analysis process is detailed in Chapter 5. Finally, Chapter 6 presents conclusions.

2. DATA DEPENDENCE ANALYSIS BACKGROUND

Data dependence analysis is a very broad topic which is summarized here for the reader's understanding. A more detailed discussion of the concepts introduced here can be found in the following references: [3], [4], [5], [6].

2.1 Data Dependence Analysis Concept

Data dependences are relationships between two accesses to the same memory location in a program and usually indicate the order in which these accesses must be executed for correct program results. A *data dependence* exists from memory reference **A** to memory reference **B** if the references access the same memory location and there exists a possible execution path through the program from reference **A** to reference **B**. Table 2.1 shows the four possible data dependence types depending on whether the references to **A** and **B** are read accesses or write accesses. Also shown in Table 2.1 is the symbolic representation of the dependence relation denoted by the symbol δ.

Table 2.1: Data Dependence Relation Types.

Reference **A**	Reference **B**	Dependence Type	Dependence Relation
write	read	Flow	**A** δ^f **B**
write	write	Output	**A** δ^o **B**
read	write	Anti	**A** δ^a **B**
read	read	Input	**A** δ^i **B**

Flow dependences are the only "true" dependences in the sense that they represent the flow of information from a write to a read of the same memory location. Output and anti-dependences are due only to the reuse of memory locations present in imperative, sequential languages such as *Fortran* and *C*. Since these dependences can always be eliminated in a semantically valid way by introducing new variables into the program, they are sometimes referred to as *artificial dependences*. Program transformations need not consider input dependences in order to insure that correct program results are obtained for Von Neumann computers. Input dependences do, however, impart information on the temporal order of read references to the same memory location, which proves useful for some memory optimizations such as *scalar replacement* [7].

2.2 Value-Based and Memory-Based Data Dependences

The definition of data dependence given in Section 2.1 is that of a *memory-based data dependence*. Memory-based dependences indicate that the associated pair of references access the same memory location, but imply nothing about the values which are accessed. *Value-based data dependences*, on the other hand, indicate that the same value in memory is accessed by both references involved in the dependence. Thus, the definition of data

dependence is extended in the following manner: A value-based data dependence exists from reference **A** to reference **B** if, and only if, a memory-based data dependence exists from **A** to **B**, and the memory location accessed by references **A** and **B** is not modified along any possible execution path in the program from reference **A** to reference **B**.

For scalar variables, value-based dependences are easily identified using a *reaching definitions and uses data flow analysis* [8]. Memory-based dependences can then be calculated by taking the transitive closure of the set of value-based dependences. In contrast, determination of value-based dependences for array variables requires more sophisticated and computationally expensive analysis techniques [9], [10].

While many program optimizations, such as code scheduling, require only memory-based dependences for validity checking, some require value-based dependences. Determination of value-based dependences are necessary for optimizations such as *scalar replacement* [7], which require knowledge of the flow of values from one reference to another.

2.3 Loop-Carried Data Dependences

To insure the validity of a loop transformation, the compiler designer often requires information about which loop iterations of the loops in a nest are involved in the dependence relations for references in the loop nest. Specifically, data dependences which occur from a reference of a loop iteration to another reference in another iteration are called

loop-carried dependences. If the loop iterations may be reordered by a program transformation, the iteration ordering specified by the loop-carried data dependences must be respected to insure the validity of the transformation. Data dependences which do not span more than one iteration of any loop enclosing both references are termed *non-loop-carried* dependences. Data dependences between both scalar references and array references may be of either type: loop-carried or non-loop-carried.

2.3.1 General loop-carried dependence analysis problem

In this section, we describe a general loop-carried data dependence analysis problem for a loop nest and show how the system of equations and inequalities for solving this problem is determined.

In Figure 2.1, a fragment of code is given to describe the data dependence analysis problem in general. Although most of the language constructs in the code fragment follow *Fortran* syntax, array reference subscripts are enclosed in square braces, following *C* syntax, to ease the identification and separation of the subscripts. Only the code relevant to the analysis problem is shown — other loops and statements may exist between the lines shown. The code fragment is a nest of loops containing two references to the same m-dimensional array, a. All capitalized variables and functions are important in defining the data dependence analysis problem. Note that the subscripts on the index variables indicate how deeply the associated loops are nested. Only reference **A** is nested within the loops with index variables $I_{n+1}, \ldots, I_{n+p}$, and only reference **B** is nested

do $I_1 = L_1,\ U_1$
  ⋱
    do $I_n = L_n(I_1, \ldots, I_{n-1}),\ U_n(I_1, \ldots, I_{n-1})$

      do $I_{n+1} = L_{n+1}(I_1, \ldots, I_n),\ U_{n+1}(I_1, \ldots, I_n)$
        ⋱
          do $I_{n+p} = L_{n+p}(I_1, \ldots, I_{n+p-1}),\ U_{n+p}(I_1, \ldots, I_{n+p-1})$

reference **A**: $a[S_1(I_1, \ldots, I_{n+p})] \ \ldots \ [S_m(I_1, \ldots, I_{n+p})]$

          enddo I_{n+p}
        ⋰
      enddo I_{n+1}

      do $I^{'}_{n+1} = L^{'}_{n+1}(I_1, \ldots, I_n),\ U^{'}_{n+1}(I_1, \ldots, I_n)$
        ⋱
          do $I^{'}_{n+p^{'}} = L^{'}_{n+p^{'}}(I_1, \ldots, I_n,\ I^{'}_{n+1}, \ldots, I^{'}_{n+p^{'}-1}),\ U^{'}_{n+p^{'}}(I_1, \ldots, I_n,\ I^{'}_{n+1}, \ldots, I^{'}_{n+p^{'}-1})$

reference **B**: $a[S^{'}_1(I_1, \ldots, I_n,\ I^{'}_{n+1}, \ldots, I^{'}_{n+p^{'}})] \ \ldots \ [S^{'}_m(I_1, \ldots, I_n,\ I^{'}_{n+1}, \ldots, I^{'}_{n+p^{'}})]$

          enddo $I^{'}_{n+p^{'}}$
        ⋰
      enddo $I^{'}_{n+1}$

    enddo I_n
  ⋰
enddo I_1

Figure 2.1: Code Fragment for Loop-carried Dependence Analysis Problem.

within the loops with index variables $I^{'}_{n+1},\ \ldots,\ I^{'}_{n+p^{'}}$. Both references are nested within the loops with index variables $I_1,\ \ldots,\ I_n$. Each array subscript for each reference and each upper and lower loop bound is a function of the enclosing-loop index variables.

Since each reference is accessed multiple times during the execution of the loop nest, each access instance is uniquely identified by a vector of the values of the enclosing-loop indices called the *iteration vector*. Suppose that the iteration vectors for references **A** and

B are defined as $\vec{i} \equiv \langle i_1, \ldots, i_{n+p} \rangle$ and $\vec{i}^{'} \equiv \langle i_1^{'}, \ldots, i_{n+p'}^{'} \rangle$, respectively, such that the lower-case letters represent specific instances of the corresponding index variable. Note that the access instance values for the outermost n loops in the nest may be different for the two references, which is represented by the $i_1, \ldots, i_n$ values for reference **A**, and the $i_1^{'}, \ldots, i_n^{'}$ values for reference **B**.

To determine the specific instances of reference **A** and reference **B** that are involved in a dependence from reference **A** to reference **B**, the following constraints are imposed to generate a system of equations and inequalities:

1. The instance $\vec{i}$ of reference **A** must occur temporally before the instance $\vec{i}^{'}$ of reference **B** during program execution.

2. Each element of $\vec{i}$ and $\vec{i}^{'}$ must be within the range of its corresponding loop bounds.

3. The element of array a accessed by instance $\vec{i}$ of reference **A** must be the same element of array a accessed by instance $\vec{i}^{'}$ of reference **B**.

These constraints can be formulated mathematically as

1. $\langle i_1, \ldots, i_n \rangle \preceq^1 \langle i_1^{'}, \ldots, i_n^{'} \rangle$, or equivalently,[2] $\vec{i}_{(1:n)} \preceq \vec{i}^{'}_{(1:n)}$

2. $L_h(\vec{i}_{(1:h-1)}) \leq i_h \leq U_h(\vec{i}_{(1:h-1)}), \ \forall h \in [1 : n{+}p] \ \bigwedge$

   $L_h^{'}(\vec{i}^{'}_{(1:h-1)}) \leq i_h^{'} \leq U_h^{'}(\vec{i}^{'}_{(1:h-1)}), \ \forall h \in [1 : n{+}p^{'}]$

[1] The symbol "$\preceq$" represents the lexicographic ordering relation. This relation is parallel to dictionary ordering if each letter corresponds to a vector element value.

[2] Only loop indices common to both references determine execution ordering of reference **A** instances with respect to reference **B** instances.

3. $S_h(\vec{\imath}) \;=\; S_h^{'}(\vec{\imath}^{'}),\;\; \forall h \in [1:m]$

The system of equations and inequalities generated above can sometimes be solved for $\vec{\imath}$ and $\vec{\imath}^{'}$ over the set of integers using *integer programming* techniques if all the S array subscript functions and L and U loop bound functions are linear functions of the loop indices. This set of solutions completely enumerates the data dependences involving these two references within the loop nest, such that $\vec{\imath}$ provides the loop index values at reference **A**, which is the source of the data dependence, and $\vec{\imath}^{'}$ provides the loop index values for reference **B**, which is the destination.

2.3.2 Specific example problem solution

The concepts just explained in Section 2.3.1 are more easily grasped through the solution of a specific example. The code fragment shown in Figure 2.2 is a triply nested loop with constant loop bounds containing two array references. We wish to determine the pairs of iteration vectors for which a data dependence from reference **A** to reference **B** exists.

The iteration vectors for references **A** and **B** are $\vec{\imath} \equiv \langle i_1,\; i_2,\; i_3 \rangle$ and $\vec{\imath}^{'} \equiv \langle i_1^{'},\; i_2^{'},\; i_3^{'} \rangle$, respectively. The three constraints described earlier generate the following set of equations and inequalities involving $\vec{\imath}$ and $\vec{\imath}^{'}$:

1. $i_1 < i_1^{'} \;\bigvee\; (i_1 = i_1^{'} \bigwedge i_2 < i_2^{'}) \;\bigvee\; (i_1 = i_1^{'} \bigwedge i_2 = i_2^{'} \bigwedge i_3 < i_3^{'})$

2. $1 \leq i_1, i_1^{'} \leq 10 \;\bigwedge\; 1 \leq i_2, i_2^{'} \leq 10 \;\bigwedge\; 1 \leq i_3, i_3^{'} \leq 10$

3. $i_1 + 1 = i_1^{'} \;\bigwedge\; i_2 = i_2^{'} \;\bigwedge\; i_3 - 1 = i_3^{'}$

do $I_1 = 1,\ 10$

do $I_2 = 1,\ 10$

do $I_3 = 1,\ 10$

reference **A**: $a[I_1 + 1][I_2][I_3 - 1]$

reference **B**: $a[I_1][I_2][I_3]$

enddo I_3

enddo I_2

enddo I_1

Figure 2.2: Code Fragment for Dependence Problem Example.

The solution set for the dependence analysis problem can be expressed in terms of pairs of iteration vectors of the form $(\vec{i},\ \vec{i}')$, such that the iteration vectors represent the source and destination of the data dependence, respectively. For this specific example, the solution set can be expressed as follows:

$$\mathcal{S} \ = \ \{\ (\langle i_1,\ i_2,\ i_3\rangle,\ \langle i_1 + 1,\ i_2,\ i_3 - 1\rangle)\ :\ 1 \leq i_1 \leq 9\ \wedge\ 1 \leq i_2 \leq 10\ \wedge\ 2 \leq i_3 \leq 10\}$$

Often, the solution set will contain multiple iteration vector pairs with a specific pattern similar to the one given above. However, much more complicated solution sets are certainly possible and do arise in practice.

2.3.3 Data dependence abstractions

Although the method of describing a loop-carried data dependence using the exact solution set yields the most specific information, more compact abstractions of the data

dependence information often prove adequate for most optimizing compiler applications. Furthermore, representing the exact solution set can be prohibitively expensive in cases where the solution set is not uniform enough to represent compactly, as it is in the dependence problem example presented in Section 2.3.2. Finally, direct enumeration of all possible solutions may not be possible since the solution set may be infinite due to symbolic loop bounds. In addition, as the level of information contained in the abstraction increases, the time complexity of the analysis algorithm tends to increase. Sometimes direct enumeration of a finite solution set is also prohibitively expensive in terms of the execution time of the analysis algorithm.

The most common data dependence abstractions are *distance vectors* and *direction vectors*, which are detailed enough to enable many transformations and optimizations, but require very little space to represent. Often, the iteration vectors of a pair in the data dependence problem solution have a constant vector difference, which is the case for the solution presented in Section 2.3.2. These data dependences are called *uniform data dependences*, and can be represented compactly by the vector difference of the iteration vectors. More precisely, the distance vector (or *difference vector*) elements for a given data dependence are calculated by subtracting an element in the source iteration vector from the corresponding element in the destination iteration vector, for all elements of the iteration vectors representing loops enclosing both references. For the code fragment given in Figure 2.1, the distance vector is defined as follows:

$$\vec{D} \equiv \vec{i}'_{(1:n)} - \vec{i}_{(1:n)}$$

This abstraction is called a *distance vector* because it represents the "distance" in terms of loop index values between the source and destination access instances for all loops which enclose both references involved in the dependence.[3] For the example dependence problem given in Figure 2.2, $\vec{D} = \langle 1, 0, -1 \rangle$, and the data dependence relation is specified mathematically as $\mathbf{A}\ \delta_{\langle 1,0,-1 \rangle}\ \mathbf{B}$.

In the case of *non-uniform data dependences*, distance vectors become impractical to store because a single dependence may be represented by many different vectors. Fortunately, a further abstraction of the distance vector called the *direction vector* can represent sets of distance vectors compactly. For many transformations, the important information is the arithmetic sign of the difference of the iteration vectors.

The direction vector elements for a given data dependence are calculated by determining the sign of the result of subtracting an element in the source iteration vector from the corresponding element in the destination iteration vector, for all elements of the iteration vectors representing loops enclosing both references. For the code fragment given in Figure 2.1, the direction vector is defined as follows:

$$\vec{d} \equiv \sigma(\vec{i}'_{(1:n)} - \vec{i}_{(1:n)}) \quad | \quad \sigma(j) \equiv \begin{cases} +, & \text{if } j > 0 \\ 0, & \text{if } j = 0 \\ -, & \text{if } j < 0 \end{cases}$$

Therefore, the direction vector's elements are elements of the set $\{+, 0, -\}$. For the example dependence problem given in Figure 2.2, $\vec{d} = \langle +, 0, - \rangle$, and the data dependence relation is specified mathematically as $\mathbf{A}\ \delta_{\langle +,0,- \rangle}\ \mathbf{B}$.

---

[3] Actually, this is only one possible definition of dependence distance. For loops which are not normalized to have an index increment of one, several different definitions are useful [11].

The data dependence analysis algorithm need not determine distance vectors in order to calculate direction vectors. In fact, some algorithms determine if solutions can exist to the set of equations and inequalities when relations are added to constrain the problem to a specific direction vector. If solutions can exist, then that direction vector is added to the solution set. Furthermore, direction vectors can be represented compactly as bit vectors with three bits per direction vector element, which represent all possible subsets of the set $\{+, 0, -\}$. This allows direction vectors to be combined in such a way that a single vector element can represent more than one possible direction. More precisely, elements of the set, $\{+, 0, -, 0+, 0-, +-,$[4] $*\}$, represent possible direction combinations for extended direction vectors, such that:

$$\begin{aligned} 0+ &\equiv (0 \vee +), \\ 0- &\equiv (0 \vee -), \\ +- &\equiv (+ \vee -), \\ * &\equiv (+ \vee 0 \vee -) \end{aligned}$$

For example, the extended direction vector, $\langle +,\ 0+,\ * \rangle$, represents the following set of simple direction vectors:

$$\{\langle +,\ 0,\ + \rangle,\ \langle +,\ 0,\ 0 \rangle,\ \langle +,\ 0,\ - \rangle,\ \langle +,\ +,\ + \rangle,\ \langle +,\ +,\ 0 \rangle,\ \langle +,\ +,\ - \rangle\}$$

Many other data dependence abstractions exist. Those presented in this section are some of the most popular, since they represent a good balance of the opposing

[4] It is difficult to imagine how this combination might occur in practice. However, since it can be represented with no loss of compactness, it is included for completeness.

properties of preciseness and compactness. A more detailed discussion of data dependence abstractions and the transformations they enable is found in [12].

3. *IMPACT* COMPILER OVERVIEW

The *IMPACT* compiler is a prototype compiler for *C* and *Fortran* programs with special language and optimization support for compiling programs for multiple-issue processors and shared-memory multiprocessors. *IMPACT* compiles code for several commercial microprocessors, as well as experimental superscalar and VLIW architectures. A block diagram showing the organization of the *IMPACT* compiler is presented in Figure 3.1.

The compiler is divided into three primary modules based on the level of the intermediate representation (*IR*) used. The *Pcode* module employs a high-level *IR* consisting of an abstract syntax tree (or *AST*) representation of the *C* language. The medium-level *IR*, employed by the *Hcode* module, is based on a control flow graph data structure but expressions are represented as *AST*s [13]. Finally, the *Lcode* module utilizes a generalized register-transfer language similar in structure to most load/store-processor instruction sets. In addition, several front-end compiler modules perform the tasks of language conversion, language preprocessing, and *IR* generation. All of these compiler modules as well as the supported architectures are described in more detail in the sections following.

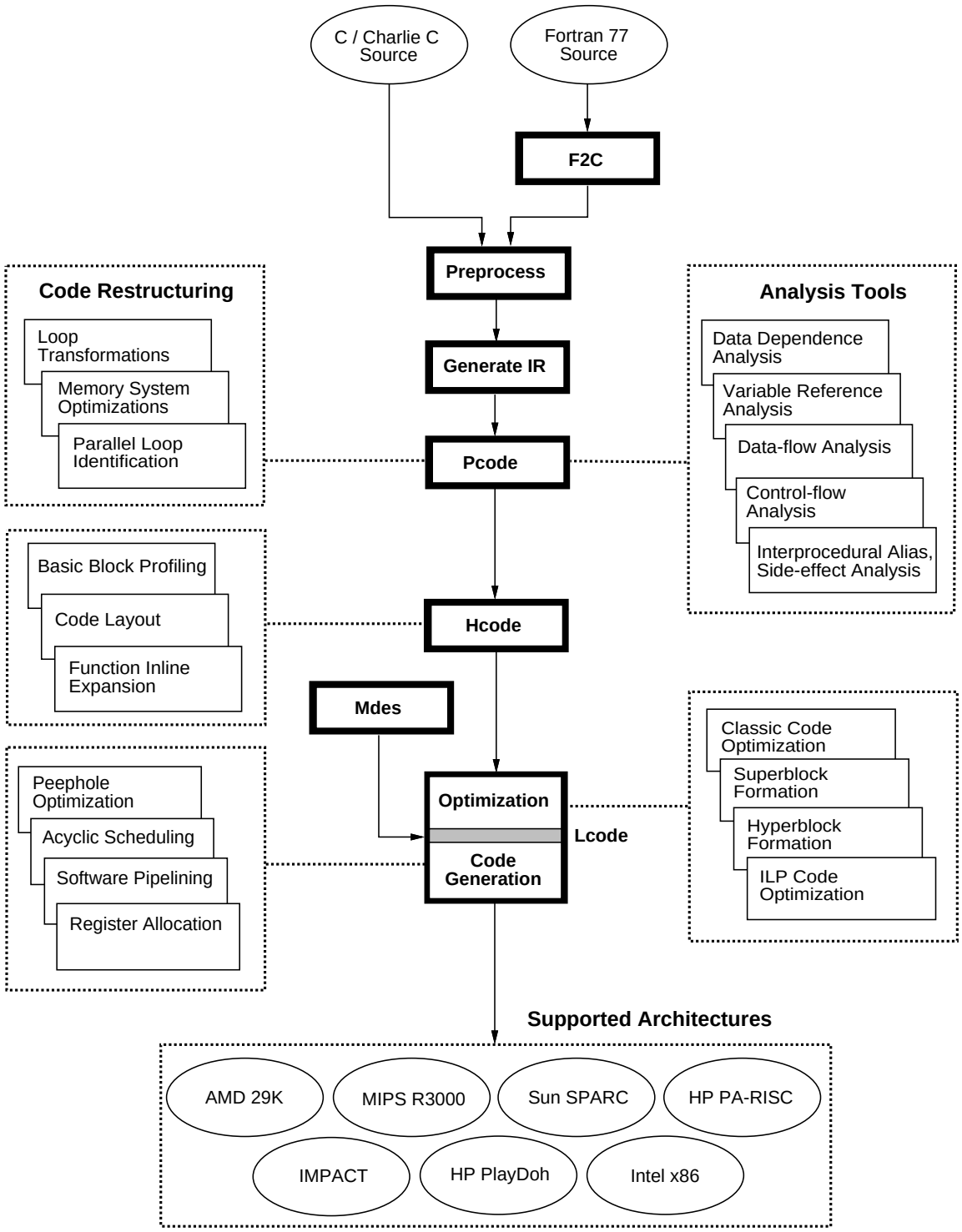


Figure 3.1: Organization of the *IMPACT* Compiler.

3.1 The Front-End Compiler Modules

IMPACT is capable of compiling both *C* [14], [15] and *Charlie C* languages directly. The *Charlie C* language is a block-parallel *C* language extension that supports loop-level parallelism as well as microtasking [16]. Compilation of *Fortran 77* [17] source is accomplished by first translating it to *standard C* via the program *f2c* [18]. The *preprocessing* module possesses *C* include file and macro preprocessing capabilities as well as facilities to annotate *f2c*-translated source code to recover array dimension information which would otherwise be lost in the *f2c* translation process. Finally, the last front-end compiler module parses *C* or *Charlie C* and generates the *Pcode IR*.

3.1.1 The *f2c Fortran* to *C* language translator

Since the *IMPACT* compiler intermediate representations are based on the *C* programming language, *Fortran* programs cannot be directly compiled without modifying the syntax and semantics of the *IR*. Although modification of the intermediate representations is perhaps a better long-term strategy, we decided to use a public domain *Fortran* to *C* language translator in order to be able to compile *Fortran* code in the short-term. However, *f2c* obscures the variable reference and aliasing convention semantics of the *Fortran* source code. Therefore, to facilitate accurate dependence analysis and effectively optimize the translated code, the *IMPACT* compiler recovers as much as possible of the original *Fortran* semantics.

One syntactic change that *f2c* makes is to *linearize* all multidimensional arrays to one dimension, since array access semantics differ between *Fortran* and *C*. In *Fortran*, multidimensional arrays are stored in *column-major* order in memory, while in *C*, they are stored in *row-major* order. When a multidimensional array is also accessed via a single-dimension address calculation, the semantics of array layout dictate which array element is accessed. To prevent incompatibilities between the *Fortran* and *C* array layout semantics, *f2c* converts all multidimensional arrays to single-dimensional arrays which are accessed using explicit address calculations.[1]

3.1.2 *IMPACT* preprocessing

For data dependence analysis to be effective, multidimensional array references are *delinearized* by the *Pcode* module, making the original array reference subscripts available for the analysis. However, for local and global arrays, the size of each array dimension must be known in order for delinearization to proceed correctly. *F2c* places the original array dimension-size information in *C* language comments after the corresponding array declaration. This dimension size information must be preserved down to the *Pcode* module, which necessitates a preprocessing module to convert the comments into *Charlie C* language *pragmas* before they are destroyed by the *C* language preprocessor.

---

[1] Actually, arrays are linearized for another reason also. According to the *standard C* language definition, the declarations of formal function parameters which are n-dimensional arrays must specify the last $n-1$ dimension sizes as integer constant expressions (which can be evaluated at compile time). However, *Fortran* allows all dimension sizes to be declared as variables (which are usually also formal function parameters). Linearizing multidimensional arrays easily circumvents this *C* language restriction.

The *IMPACT* preprocessing module accomplishes the usual *C* language source preprocessing while preserving the array dimension size information generated by *f2c*. First, the source file is preprocessed by the target machine *C* compiler to expand macros and included source files, but comment deletion is suppressed. Next, local and global variable declarations as well as global structure definitions are scanned for the presence of comments of the form: `/* was [<dim_1_size>][<dim_2_size>]...[<dim_n_size>] */`. The comments are then converted to delinearization pragmas which are placed just before the variable declaration occurs in the source file. These pragmas contain the original array dimensions and a string which represents the complete variable access string textually (including struct and/or union fields). This string is necessary to match pragmas to the associated array since *Fortran* common blocks are translated to *C* **structure** constructs that contain the common block arrays as fields (see Section 4.1.1), and pragmas may only be attached to entire variable declarations — in this case, structure variable declarations. Finally, the source file is again preprocessed by the target machine *C* compiler to remove all comments.

One other function the *IMPACT* preprocessor performs is to identify the source language of the file (either *C* or *Fortran*) so that the *Pcode* module can appropriately analyze the variable references according to the source language semantics. Other syntactic code analyses and transformations can be added easily to the *IMPACT* preprocessing module since it is written in the report-generation language *perl*.

3.1.3 Intermediate representation generation

The intermediate representation generation module for the *IMPACT* compiler converts preprocessed *C* or *Charlie C* code into the *Pcode* intermediate representation. In addition, this module performs limited syntactic and semantic checking and attaches each *Charlie C* pragma to the lexically nearest function, global variable, local variable, or program statement (depending on the pragma type) which follows the pragma.

3.2 The *Pcode* Compiler Module

The *Pcode* compiler module performs high-level analyses, transformations, and optimizations which benefit from explicit source-level information. Furthermore, the *Pcode* intermediate representation facilitates the manipulation of hierarchical program structures such as loops and blocks of statements. The following sections discuss the intermediate representation, analysis tools, and transformations and optimizations present in the *Pcode* compiler module.

3.2.1 Intermediate representation

The highest-level *IR*, *Pcode* [19], is an abstract syntax tree which closely mimics the nested-statement syntax of the *Charlie C* language. The tree contains three kinds of nodes: *function*, *statement*, and *expression*.

Each function node contains information specific to a *C* language function such as a list of formal parameters, function pragmas, and the type of data returned by the

function, as well as a pointer to the compound statement node representing the body of the function.

Statement nodes represent *C* statements, and may contain pointers to children statement nodes, representing the nested statement structure of *C*. The type of the statement determines what type of information is present in the statement node as well as the node type (statement or expression, but not function) and number of its child nodes. A compound statement node, for example, represents the *C* curly-brace construct and contains a list of the local variables declared within its scope as well as a pointer to a list of statements contained lexically within it.

An expression node represents a *C* language expression such as a function call or variable assignment. These expression nodes are used to represent all *C*-language operators, numeric constants, and variables. Again, the type of the expression node determines the specific information present in the node as well as the number of expression node children it possesses. Expression nodes may not contain statement or function nodes as children.

A symbol table is maintained at the level of the source function and contains variable, structure, union and enumeration declaration information. Local variable names are appended with a unique lexical scope identifier to insure they have unique names in the symbol table used for the entire function. Therefore, symbol tables are not necessary for each scope in a source code function.

An additional integral component of the *Pcode IR* is the *control flow graph*, which is composed of blocks of straight-line code called *basic blocks*. Directed arcs representing

the flow of control of the program connect the basic blocks to form a directed, cyclic control flow graph for a function. Each basic block is subdivided into one or more *flow nodes*, which represent units of code which execute sequentially. Expression statements are represented by a single flow node in the control flow graph, while complex statements, such as loops, are represented by several flow nodes. Pointers exist from the statement nodes to the associated flow node(s) and from each flow node back to its associated statement node. Expression-level flow of control, such as that present in conditional expressions and short-circuit evaluation operators, is not represented in the control flow graph.

The control flow graph representation is utilized for data flow analysis, unstructured loop detection and nesting determination, and data dependence analysis. Since the control flow information is utilized by nearly every *Pcode* module functionality, the control flow graph is maintained as part of the intermediate representation. However, since it can be easily and efficiently constructed, the control flow graph is not modified after every code transformation, but is instead reconstructed at the end of each transformation.

3.2.2 Analysis tools

The *Pcode* module analysis tools are utilized by nearly every code restructuring optimization and transformation, to insure their validity and efficacy. Furthermore, many of the analysis tools are used by other analysis tools in a layered manner.

Control flow analysis is perhaps the most important analysis tool since nearly all other analysis tools utilize it. It consists of control flow graph construction, loop detection and nesting determination (used mostly for unstructured loops), and non-loop-carried reachable control flow analysis used by data dependence analysis (see Section 5.3.1). During control flow graph construction, unreachable code is removed from the *Pcode IR*, which is necessary for some data flow analyses to function correctly.

The control flow graph also provides a structural framework on which to build data flow information. Data flow analysis determines the flow of program values, variables, and expressions throughout the control flow graph. Traditional types of data flow information are computed including sets of reaching definitions and uses, available definitions and uses, and live variables [8], for each basic block and/or flow node in the control flow graph.

The data flow analyses are performed by request from other analyses or transformations, and the resulting information, represented as sets, is stored in the flow nodes and basic blocks. The convention for preventing the use of stale information in the *Pcode* module is to remove all information that a particular *IR* modification could render incorrect. Therefore, other analyses or transformations can safely utilize any data flow information provided that it is still available in the control flow graph. Data flow information is also removed automatically from the control flow graph whenever the variable access table (explained below) is destroyed.

The next two analysis tools, variable reference analysis and data dependence analysis, are the subject of this thesis and are explained in detail in the succeeding chapters. Variable reference analysis builds a variable access table containing information for each distinct variable reference in the source function. Pointers between each entry in the variable access table and the associated variable access expression in the abstract syntax tree are provided. Because the variable access table contains general information about the variable references, it is utilized by not only data dependence analysis, but also data flow analysis and other transformations and optimizations.

Data dependence analysis calculates dependence distance and direction vectors for all data-dependent pairs of variable accesses in the function. The resulting data dependences are represented as annotated arcs between pairs of entries in the variable access table, forming a *data dependence graph* for the entire function. Options passed to the data dependence analyzer determine whether dependences are to be calculated for pairs of variables with no common loops and whether non-loop-carried dependences are to be calculated between variable accesses within the same expression statement. This allows the transformation that uses the data dependence information to tailor the scope and granularity of the information to its specific needs.

As with data flow information, dependence information must be destroyed if it is rendered incorrect by a program transformation. The easiest way to prevent use of stale data dependence or data flow information is to destroy the variable access table. We have found that rebuilding the data dependence information after each program

transformation is a reasonably efficient strategy for a prototype compiler. Although incremental update of data dependence information is possible, it is currently a topic of ongoing research, and is not supported by the *Pcode* module.

Finally, interprocedural alias and side-effect analysis generates a list of aliased variables and side effects for function calls [20]. Data dependence analysis uses this information to determine variable aliasing and function call side effect relationships, which is explained in Section 4.2.

3.2.3 Transformations and optimizations

The *Pcode* module contains several code restructuring transformations and optimizations which utilize the analysis tools frequently.

General purpose loop transformations currently implemented include loop distribution or loop fission (with statement reordering), loop interchange (for loop nests with rectangular iteration spaces), loop skewing, and loop reversal [21]. These loop transformations are typically exploited as tools to improve the applicability of other transformations and optimizations.

Memory system optimizations include loop blocking (also called iteration space tiling) to improve cache access locality [22], software prefetching, and *data relocation and prefetching* [23], a hardware-assisted form of software prefetching which simultaneously relocates array data to reduce cache mapping conflicts.

Parallel loop identification is currently limited to loops which can be software pipelined in the *Lcode* module. Aggressive automatic loop parallelization for shared-memory multiprocessors would require implementation of data dependence breaking transformations and other supporting transformations and analyses. Multiprocessor loop parallelization has not been a major focus of the *IMPACT* compiler research group up to this point in time.

3.3 The *Hcode* Compiler Module

The *Hcode* module performs profiling at the level of the basic block. Additionally, profile-guided code layout and function inline expansion are performed by the *Hcode* module [24], [25]. Eventually, all the functionality provided by this module will be relocated to the *Pcode* and *Lcode* modules. For instance, function inline expansion will be moved to the *Pcode* module in order to apply high-level optimizations across function call boundaries.

3.4 The *Lcode* Compiler Module

The final module in the *IMPACT* compiler is referred to as the *Lcode* module. Using the *Lcode IR*, all machine-independent classic optimizations are applied [26]. Superblock [27] and hyperblock [28] compilation techniques are also performed using the *Lcode IR*.

All code generation in the *IMPACT* compiler is also performed using the *Lcode* module. Scheduling is performed via either acyclic global scheduling [29], [30] or software pipelining using modulo scheduling [31]. Graph-coloring-based register allocation is utilized for all target architectures [32]. In addition, for each target architecture, a set of specially tailored peephole optimizations are performed.

A detailed machine description database, *Mdes*, for the target architecture is also available to all Lcode compilation modules [33].

3.5 Architectures Supported by the Compiler

Several architectures are supported by the *IMPACT* compiler. These include the *AMD 29K* [34], *MIPS R3000* [35], *Sun SPARC* [36], *HP PA-RISC*, and *Intel x86*. The other two supported architectures, *IMPACT* and *HP Labs' PlayDoh* [37], are experimental architectures incorporating instruction-level parallelism.

4. VARIABLE REFERENCE ANALYSIS

The data dependence analyzer in the *IMPACT* compiler relies on detailed analysis of the program variable references. This reference analysis recovers the variable reference semantics of the original *Fortran* source code and gathers detailed information about referenced variables and their relationships. Since other analysis tools, optimizations, and transformations use much of this variable reference information, reference analysis is a distinct phase of the *Pcode* module which may be invoked independently of data dependence analysis. However, most of the variable reference preparation for data dependence analysis is performed during the reference analysis phase. The information gathered by reference analysis is stored in a data structure called the *access table*.

4.1 Access Table Data Structures

The access table consists of a hierarchy of data structures that organize the variable accesses within a function, as shown in Figure 4.1. At the highest level of the hierarchy, each variable referenced in the function is represented by an entry in the access table,

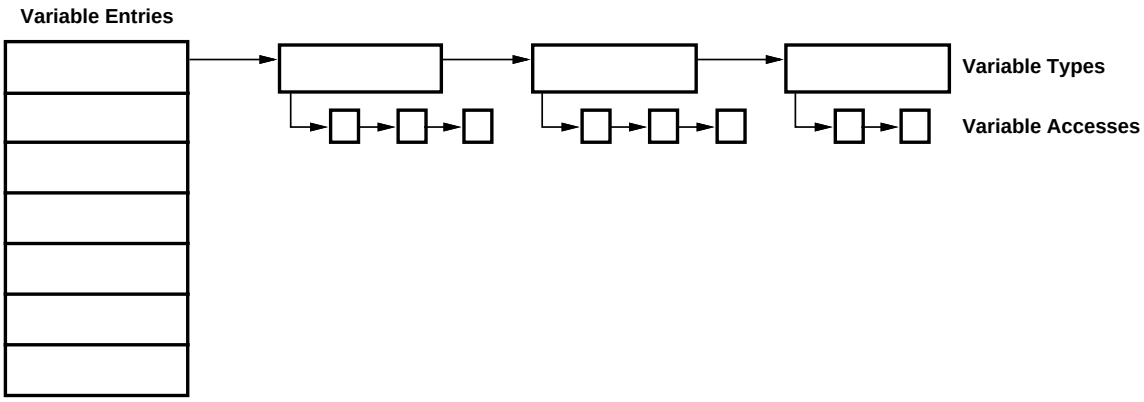


Figure 4.1: Organization of the Access Table Data Structures.

called a *variable entry*. Within each variable entry is a list of *variable types*, which represent the different data types used when referencing the variable in the function. *Fortran* **equivalence** constructs create the need for distinct variable types, which will be explained in more detail in Section 4.1.2. Each variable type contains a list of *variable access* structures, which consist of information specific to a particular variable reference in the source function.

To build the access table, each variable reference in the program is separately analyzed and the resulting information placed in the appropriate data structures in the access table. New data structures in the access table are allocated as needed during this process. Note that variables which are never referenced in the function being analyzed are not allocated a variable entry in the access table. Each of the access table data structures is explained in more detail in the following sections.

4.1.1 Variable entry data structure

The access table is implemented as a hash table, keyed on the variable name. The *f2c*-translation process sometimes places groups of *Fortran* code variables in the same *C* code aggregate variable. Consequently, the access table differs from the *Pcode* symbol table in that a variable entry exists for each distinct array and scalar variable referenced in the original *Fortran* source code. Furthermore, the symbol table contains information about the variable declarations, whereas the access table contains information about the actual variable references.

F2c converts *Fortran* **common blocks** and variables of type *complex* (as shown in Figure 4.2(a)) to *C*-language **structures** (as shown in Figure 4.2(b)). Therefore, a variable reference in the original *Fortran* code is sometimes represented in the *C* code by a reference to a field of a structure. Furthermore, as shown in Figure 4.2(b), references may occur to both an entire complex variable (**cb.c**) and the real and/or imaginary parts of the same variable (**cb.c[0].i** and **cb.c[0].r**). For this reason, structure element references and references to entire structures are assigned unique variable entries in the access table. The hash key for the variable entry is a string consisting of the (aggregate) variable name appended to a dot-separated-list of the structure field names present in the reference of that variable. The presence of array indexing in the variable references has no effect on the variable entry hash keys themselves since references to the same array are associated with the same variable entry data structure. For example, in Figure 4.2, references to the array of complex values, such as **cb.c.[1].r**, are grouped in the access

```
COMMON /CB/ C, D, R, M
COMPLEX C(10)
DOUBLE PRECISION D(10)
REAL R(10)
LOGICAL M(10)

IF (M(1)) C(1) = C(2) + C(3)

CALL FOO(C(1), R(2), D(1))
STOP
```

(a)

```
typedef struct {
    float r, i;
} complex;
struct {
    complex c[10];
    double d[10];
    float r[10];
    int m[10];
} cb;
main() {
    complex q;

    if (cb.m[0]) {
        q.r = cb.c[1].r + cb.c[2].r,
        q.i = cb.c[1].i + cb.c[2].i;
        cb.c[0].r = q.r,
        cb.c[0].i = q.i;
    }
    foo(cb.c, &cb.r[1], cb.d);
}
```

(b)

Figure 4.2: *F2c* Translation of Common Blocks and Complex Variables.

table under the variable entry with hash key **cb.c.r**, as if they were actually referencing parallel arrays of real or imaginary values, e.g., **cb.c.r[1]**.

As an example, the hash keys for the variable entries for the code in Figure 4.2(b) are the following: **cb.c**, **cb.c.i**, **cb.c.r**, **cb.d**, **cb.m**, **cb.r**, **q.i**, **q.r**,[1] and **foo**. Note that function calls are also assigned a variable entry in the access table, in order to represent the variable references implicit in function call side effects. Further details are provided in Section 4.2.3.

[1]The variable **q** is an *f2c*-generated temporary that enforces proper order of evaluation for expressions involving complex values.

```
DOUBLE PRECISION D(10)
REAL R(20)
EQUIVALENCE (D(1), R(1))

D(2) = 1.0
R(3) = 0.0
STOP
```

(a)

```
main()
{
    double equiv[10];

    equiv[1] = 1.0;
    ((float *)equiv)[2] = 0.0;
}
```

(b)

Figure 4.3: *F2c* Translation of Equivalenced Arrays of Different Type.

The access table variable entries contain information regarding the variables referenced in the function including whether each variable is local, global, a function name, or a formal parameter. Another field in the variable entry indicates whether the variable is ever referenced using an array index in the function — if not, the variable can be treated as a scalar for data flow and data dependence analyses. Other fields related to variable alias analysis are described in Section 4.2. Additionally, each variable entry contains a pointer to a linked list of variable type data structures.

4.1.2 Variable type data structure

The *Fortran* programming language provides a construct, called **equivalence**, which can be used to reference the same memory location using variables of different type. **Equivalence** constructs are often used in this manner so that an array of complex numbers may also be referenced as an array of alternating real and imaginary floating-point numbers. Another use of **equivalence** constructs allows arrays to be reused in different parts of a function, even if their element types are different. In Figure 4.3(a),

the array is referenced both as **D**, an array of double precision floating-point numbers, and as **R**, an array of single precision floating-point numbers.

Fortran equivalenced arrays are converted by *f2c* to an array of one of the types declared in the **equivalence** construct, and *C* pointer manipulation and casts are used to reference this array with different data types, as shown in Figure 4.3(b). If an array is referenced with different data types of different sizes, the corresponding array elements referenced using these two types do not reside in the same place in memory. For example, if variable data types are ignored, the two references to the **equiv** array in Figure 4.3(b) seem to access different memory locations since the array subscripts differ. However, because the type sizes of the array elements differ, the references actually access overlapping memory locations, assuming that data of type **double** requires twice as much memory to represent as data of type **float**. Variable references are grouped into separate variable type data structures based on the type to which the reference is cast to streamline the process of data dependence analysis. When comparing references to array elements of different types, the data dependence analyzer ignores the subscript expressions in order to produce conservative results.

The variable type data structure contains information describing the type of the cast if one is present. The absence of the type cast represents references of the declared variable type in the *C* source code. Also included in the variable type data structure is a linked list of variable access data structures, one for each reference to the variable

with the given type cast. Section 5.4.1 describes how this information is used to generate correct data dependence analysis results.

4.1.3 Variable access data structure

Each variable access data structure contains information specific to a single lexical variable reference in the *f2c*-translated source function. Pointers between the variable access data structure and the *Pcode* variable expression node link the reference in the *IR* and in the access table.

General information is gathered by variable reference analysis, such as whether the variable reference is a write, read, or both (as is the case for the variables which are pre/post-incremented/decremented, or for variables on the left-hand side of compound assignment operators such as "+="). For subscripted variable references, a field in the variable access data structure specifies the *extent* of the access. For instance, since all variables are passed by reference in *Fortran*, a subscripted array reference that is an actual parameter to a function implies that the function may access any element of the array by subscripting the formal parameter. In the translated *C* code, this actual parameter reference includes an address operator, "&", which implies that a pointer to an array element is passed to the function (see Figure 4.2). Consequently, data dependence analysis treats the reference as a potential access to any element of the array by disregarding the subscripting expression.

In addition, information related to the loops enclosing the reference is kept in the data structure for the purposes of dependence analysis. The loop-nesting depth of the reference is present as well as whether or not the reference is to a loop index variable in one of the enclosing loops. If the reference is to a loop index variable, the loop-nesting depth of the corresponding loop is also recorded. *Loop-nesting depth* is defined for variable references as the number of loops enclosing a given variable reference, and for a given loop in a nest, the number of loops enclosing the given loop plus one.

One other field in the variable access data structure is a pointer to a linked list of *array subscript* data structures which contain information pertaining to a single subscript for an array reference. In the *C* code, array references possess only one subscript due to the effects of *f2c* array linearization. However, for array references which were multi-dimensional in the original *Fortran* source code, the single subscript expression is split into multiple subscript expressions during *array reference delinearization*, which is the subject of Section 4.3. After array reference delinearization, each subscript expression is converted into a linear function of the loop index variables and symbolic constants, which is discussed in more detail in Section 4.4. After this array subscript processing, each array subscript data structure contains a pointer to the root of the corresponding subscript expression in the *Pcode* abstract syntax tree, as well as a field containing the linear representation of the subscript expression.

4.2 Variable Aliasing Considerations

Variable reference analysis must determine whether variables are *aliased* in order for data dependence and data flow analyses to give valid results. Variable aliasing occurs when two or more logical variables can access the same location in memory. If data dependence analysis did not take into account variable aliasing, dependences between aliased variables would not be found, and invalid code transformations may reverse the order of the references to aliased variables.

In *Pcode* variable reference analysis, aliasing relationships between referenced variables are represented by an undirected graph, such that the nodes represent variable entries in the access table, and the edges represent the possibility of variable aliasing between the variable entries they connect. The *alias graph* is constructed by comparing each pair of variable entries in the access table and connecting them with an edge if they could possibly be aliased. More precise aliasing information, such as the difference in starting addresses of aliased array variables, has not been included in the alias graph since it is usually not known at compile time.

Variables may be aliased due to explicit source language aliasing constructs or the *f2c* translation processes. In the case of source language aliasing constructs, correct construction of the alias graph ensures that any further analysis and transformations take into account the possibility that aliased variable references map to the same location in memory. Sections 4.2.1–4.2.3 describe the conditions under which such aliasing edges are added between pairs of variables in the alias graph. If the aliasing is a result of

the language translation process obscuring references to physically distinct variables, the variable reference analysis must determine which of the variables was referenced in the original source, which is called *variable disambiguation*. Section 4.2.4 describes the disambiguation of *equivalenced* variables which are aliased by *f2c* source language translation.

4.2.1 Aggregate variable aliasing

As described in Section 4.1.1, a single variable entry in the access table exists for each referenced field of a **structure** variable as well as for references to the structure variable itself (if such a reference exists). Since an access to the **structure** variable may reference the same memory location as an access to one of the fields of the **structure** variable, the entries representing these two accesses should be connected by an alias edge in the alias graph. However, accesses to different fields of the same **structure** variable should not be aliased, since these accesses reference distinct memory locations. For example, after alias graph construction, the set of pairs of aliased variable entries for the source code shown in Figure 4.2(b) is as follows: {(**cb.c**, **cb.c.i**), (**cb.c**, **cb.c.r**)}.

The condition for aggregate variable aliasing between a pair of variable entries is that one variable entry's hash key be a substring of the other variable entry's hash key, and the character following the common substring in the longer hash key be a dot. The last part of the condition is to ensure that a pair of variable entries with hash keys **var** and

varr are not aliased, but hash keys **var** and **var.r** imply that the variable entries are aliased.

Although **union** constructs are also present in *f2c*-translated source code, they are only generated when the same common block is defined differently by two or more source functions. In this case, the different common block definitions (each declared as a **structure**) are declared as fields in the **union** declaration, insuring that access to the common block by different functions reference the same block of memory. However, since references to different fields of the **union** only occur in separate source functions, there is no need to alias **union** field references unless function inlining is implemented. Implementation of **union** field reference aliasing is rather simple, and this form of aliasing also applies to variable reference analysis of *C*-language source code.

4.2.2 Function formal parameter aliasing

In the *C* language, calling a function with two actual parameters which are pointers to the same variable is not prohibited by the language definition. However, in *Fortran*, the language definition states that formal function parameters cannot be aliased, in order that optimizations may be applied more widely. Therefore, formal parameters to a function are not aliased during alias graph construction unless the programmer of the source code includes a pragma for the function describing which formal parameters are aliased.

Another *Fortran* language restriction prevents a global variable from being referenced directly in a function to which it is also passed as an actual parameter. Hence, formal

parameters are not aliased to global variables, which often results in significantly greater opportunity for code restructuring.

4.2.3 Function call side effects and aliases

Function call sites present an obstacle to code restructuring transformations because the functions called may have *side effects*, such as modification of the actual parameters, modification of global program variables, and/or access to an input/output device. If the code is restructured without regard to these side effects, incorrect program execution may occur. Although one approach to this problem is to forbid transformation of any code region which contains function calls, this approach limits the effectiveness of restructuring transformations, especially in the presence of library function calls which have no relevant side effects.

To increase the effectiveness of restructuring transformations and optimizations, variable reference analysis summarizes the effects of function call side effects in the access table and alias graph. By including function calls in the access table, as mentioned in Section 4.1.1, an alias may be created between a function call and a reference to a global variable which is also modified in the called function. In addition, function call variable entries for functions which modify the same global variable or access the same input/output device are aliased. The data dependence analyzer inserts appropriate dependences between these aliased variables; therefore, restructuring transformations need not explicitly check whether function calls have side effects, but need only check that

data dependences do not prevent valid transformation. Since input dependences do not prohibit code reordering, but all other types of dependences do, function calls with no side effects are noted as read accesses in the access table, while those for functions with side effects are noted as write accesses.

The *function information table*, which summarizes function call side effects, is queried to obtain specific side-effect information during variable reference analysis. The current implementation of the function information table includes only *f2c* library functions but may be extended to include programmer-defined functions gathered via interprocedural analysis. The variable name of the function is used as a hash key to access the specific information in the function. Currently, the function information is limited to bits specifying whether or not the function modifies any parameter, modifies any global variable, or accesses any input/output device. Therefore, if the information bits in a table entry are all zero, the function corresponding to that entry is entirely free of side effects. If the function variable name is not found in the table, all possible side effects are assumed to exist.

During access table construction, the function information table is queried to determine whether or not an actual parameter variable passed to a function may be modified during the function call. In *C*, this information could be determined by whether or not the variable or a pointer to the variable is passed as an actual parameter to the function. However, since the calling convention is *call-by-reference* in *Fortran* programs, *f2c* always passes an actual parameter variable pointer to the function. If the information returned

by the function information table indicates that the function may modify any parameter, the actual parameter variable reference is recorded as both a write access and a read access, indicating that the parameter variable could be both read and written during the function execution. If the function does not modify any parameter, the reference is recorded as a read access only.

In the absence of interprocedural side-effect analysis, aliases involving function calls are determined as follows: All variable entries for function calls which may reference any global variable are aliased to all global variable entries. Furthermore, all variable entries for function calls which may reference any global variable are aliased together, as are those entries for function calls which access input/output devices. Although these aliasing criteria appear very conservative, less conservative criteria require interprocedural analysis [20] to identify which global variables are modified and which input/output devices are accessed by each user function.

4.2.4 *Equivalenced* variable disambiguation

Fortran language programmers sometimes declare local "work" arrays which are subdivided into smaller arrays via insertion of **equivalence** constructs. Often the programmer subdivides the work array into temporary arrays whose contents are used only during the scope of a logical region[2] of the function, thus conserving memory space by allowing temporary arrays used in separate logical regions to overlap memory.

[2]For example, a logical region usually consists of a set of lexically consecutive loop nests which utilize some non-overlapping set of temporary arrays.

```
REAL A(50), B(50), C(100)
EQUIVALENCE (C(1), A(1))
EQUIVALENCE (C(51), B(1))

A(1) = 0 /* region 1 */
B(1) = 1
...
C(1) = 2 /* region 2 */
STOP
```

(a)

```
main()
{
    float equiv[100];

    equiv[0] = 0.0;
    (equiv+50)[0] = 1.0;
    ...
    equiv[0] = 2.0;
}
```

(b)

Figure 4.4: *F2c* Translation of Offset Equivalenced Arrays.

Unfortunately, *f2c*-translation effectively aliases the work array and all *equivalenced* temporary arrays. For example, Figure 4.4(a) shows a work array, **C**, which is subdivided into temporary arrays **A** and **B**. Array **A** maps to array elements **C(1:50)**, and array **B** maps to array elements **C(51:100)** in memory. The arrays **A** and **B** are utilized during the first logical region of the main program, and the array **C** is used during the second logical region. However, references to the arrays **A**, **B**, and **C** in the *Fortran* code (Figure 4.4(a)) are translated to references to the single array **equiv** in the *C* code (Figure 4.4(b)). Thus, the arrays in the original *Fortran* source are effectively aliased by the language translation process.

Since references to the **equiv** array with different offsets exist in the translated source code, these references must be partitioned by offset value in order for data dependence analysis to produce correct results. In Figure 4.4(b), for example, a simple array subscript comparison indicates that the first two references to the **equiv** array are data dependent even though they do not reference the same location in memory. Although the offset could

be incorporated into the array subscripts, the use of multidimensional arrays prohibits a general solution of this type, since array subscript linearization has serious drawbacks as will be discussed in Section 4.3. Instead, a separate access table variable entry is allocated for each offset to the referenced variable, and the separate entries are accessed using the offset value as part of the hash key. In Figure 4.4(b), two variable entries are allocated for variable **equiv**, one with offset zero and the other with offset 50. Next, all access table entries for a single array variable referenced with different offsets are aliased together during alias graph construction. Therefore, the subscripts of array references are used to prove independence of two references to the original array **A**, but all data dependences are generated between references to **A** and references to **B** regardless of array reference subscripts.

Unfortunately, this organization of the access table to ensure correctness of data dependence analysis produces overly conservative results, since a data dependence cannot exist between a reference to **A** and a reference to **B** (unless array **A** is referenced out of bounds, which is a semantic violation in *Fortran*). To *disambiguate* or *unalias* references to array **A** and array **B** in the *f2c*-translated code, the *Pcode* module includes an **equivalence** construct disambiguation parameter specifying whether array references with different offsets are to be aliased during alias graph construction. If the variable entries with different offsets are not aliased, dependence analysis treats the variable entries as non-intersecting arrays with regard to memory layout.

Unfortunately, using the "no aliasing" value for the disambiguation parameter is sometimes unsafe since references to array **B** are not aliased to references to array **C**; therefore, no data dependences are found even though they may exist between the references to the arrays. Furthermore, the *f2c*-translated code provides no information to distinguish the references to array **A** from the references to array **C**, so array **B** cannot be aliased to array **C** without also being aliased to array **A**. Fortunately, for the benchmarks compiled by *IMPACT* so far, the references to overlapping arrays occur only within separate logical regions and the code reordering transformations do not reorder entire loop nests. For several benchmarks, the use of this potentially unsafe disambiguation has proven quite effective for code optimization.

4.3 Array Reference Delinearization

Array linearization is the process of collapsing a multidimensional array into one with a single dimension. Compilers often perform this transformation before machine code generation in order to map a multidimensional array onto the computer memory, which is addressed in a manner similar to a single-dimensional array. As described in Section 3.1.1, *f2c* linearizes all multidimensional arrays in order to accommodate *Fortran* and *C* language differences in mapping multidimensional arrays to memory. In addition, programmers, as well as *C* and *Fortran* compilers, sometimes perform array linearization of source-level code [38].

```
foo (int U, float a[ ][10]) {
      int I,J;

      for (I=0; I<U; I++) {
            for (J=0; J<U; J++) {
ref A:            a[I][J] = . . . ;
ref B:            . . . = a[I][J];
            }
      }
}
```

(a)

```
foo (int U, float a[ ]) {
      int I,J;

      for (I=0; I<U; I++) {
            for (J=0; J<U; J++) {
ref A:            a[10*I + J] = . . . ;
ref B:            . . . = a[10*I + J];
            }
      }
}
```

(b)

Figure 4.5: Array Linearization Example.

Although some compilers choose to linearize all arrays in order to simplify the intermediate representation, the accuracy of data dependence analysis may decrease significantly due to the combining of array reference subscript expressions. Figure 4.5(a) shows a *C*-language function with a two-dimensional array referenced in a double-loop nest. Data dependence analysis for references **A** and **B** can easily determine that the only existing dependence is $\mathbf{A}\ \delta^{f}_{\langle 0,\,0\rangle}\ \mathbf{B}$ (using the dependence distance vector abstraction of the dependence relation).[3] Therefore, no data dependence exists between different iterations of the loops in the nest.

Figure 4.5(b) shows the same function with array **a** linearized to a single dimension. In the linearized case, the subscript expressions have been collapsed into a single expression, which affects the data dependence analysis problem constraints. The system of

[3]Here, the assumption is made that arrays must be referenced within the declared array bounds, which is an entirely reasonable assumption for *Fortran* programs.

equations and inequalities (numbered according to the constraints given in Section 2.3.1) representing the potential flow dependence in terms of iteration vectors $\langle i,\ j\rangle$ of reference **A** and $\langle i^{'},\ j^{'}\rangle$ of reference **B** are as follows:

1. $i \leq i^{'} \ \bigvee \ (i = i^{'} \bigwedge j \leq j^{'})$

2. $0 \leq i, i^{'} \leq U \ \bigwedge \ 0 \leq j, j^{'} \leq U$

3. $10i + j \ = \ 10i^{'} + j^{'}$

The solution set for this analysis problem consists of infinitely many pairs of iteration vectors as follows:

$$\mathcal{S} \ = \ \{\ (\langle i,\ j\rangle,\ \langle i+n,\ j-10n\rangle) \ :\ n \geq 0 \ \wedge \ i \geq 0 \ \wedge \ j \geq 10n\ \}$$

Since not all the distance vectors for the solution pairs are identical, the flow dependence is not a uniform data dependence and cannot be represented compactly using the dependence distance vector abstraction. The direction vector abstraction of the dependence relation is **A** $\delta^{f}_{\{\langle +,-\rangle,\ \langle 0,0\rangle\}}$ **B**, which indicates that data dependences may exist between different iterations of the loops in the nest.

The data dependence relation for the linearized array references is much less accurate than that for the original array references because two subscript expressions each containing a single loop-index variable have been combined into a single subscript expression containing both loop index variables. The false data dependences are calculated in the linearized case because different combinations of index variable values (e.g., $\langle I, J\rangle = \langle 1, 0\rangle$ and $\langle I, J\rangle = \langle 0, 10\rangle$) could reference the same array element given that the upper bound

of loop index variable J is unknown at compile time, but only one combination of index variable values may reference each element of the array in the nonlinearized case regardless of the loop index upper bounds. Although these false data dependences are calculated for linearized arrays only when certain loop index bounds are unknown at compile time, this case occurs frequently enough in *Fortran* programs to restrict the application of certain code transformations and optimizations. In addition, the efficiency of most data dependence analysis algorithms decreases as a result of array linearization. Therefore, array linearization should usually be avoided in order to increase the accuracy and efficiency of data dependence analysis.

Array delinearization is the process by which linearized single-dimensional array references are translated back to multidimensional array references. Typically, the program array references are not actually transformed, but rather the array references are delinearized for the purpose of program analyses such as data dependence analysis. In fact, actual code transformation of arrays from linearized to delinearized form may not be possible, which is the case for f2c-translated code, since multidimensional arrays may have adjustable dimensions in *Fortran*, but not in *C*. Although general algorithms to delinearize array references exist [38], they assume that array dimension sizes are unknown, making them inefficient for delinearizing array references for *f2c*-translated code. Fortunately, *f2c* leaves enough clues in the translated *C* code to simplify the process of delinearization of array references in most cases.

The next two sections describe the *IMPACT* compiler algorithm for recovering the original array subscript expressions from the *f2c*-translated code. First, Section 4.3.1 describes the process of delinearization for formal parameters. Next, Section 4.3.2 describes how to convert the problem of local and common block delinearizations to that of formal parameter delinearization.

4.3.1 Formal parameter delinearization

In *Fortran*, formal parameters that are arrays are frequently dimensioned using other formal parameters as shown in Figure 4.6(a). For array dimension sizes which are not known at compile time, the *f2c*-translated array reference subscripts contain references to "dimension" variables which hold dimension sizes computed in the function prologue, as shown in Figure 4.6(b). A *C* code array subscript expression is generated by multiplying each *Fortran* code subscript expression by the product of the sizes of the dimensions to the left of the original subscript expression and summing these products. The sum of products is then factored to remove redundant multiplications by the same dimension variable, resulting in the subscript expression for array **c** shown in Figure 4.6(b).

Since array subscript indexing begins at any integer for *Fortran* code and at zero for *C* code, *f2c* offsets the array variable pointer by enough elements to make up for the difference in all dimensions combined. The offset is calculated in an analogous manner to the linearization of array subscripts, except the subscript values used are the lower bound indices in the *Fortran* code for each dimension. However, the offset is not used

```
SUBROUTINE FOO(C, N1, N2, N3, N4)
REAL C(*)
INTEGER N1, N2, N3, N4

INTEGER I, J, K


DIMENSION C(N1, N2, 0:N3, N4)




DO K = 1, N4
   DO J = 1, N2
      DO I = 1, N1
         C(I, J, 0, K) = 0.0
      ENDDO
   ENDDO
ENDDO
RETURN
```

(a)

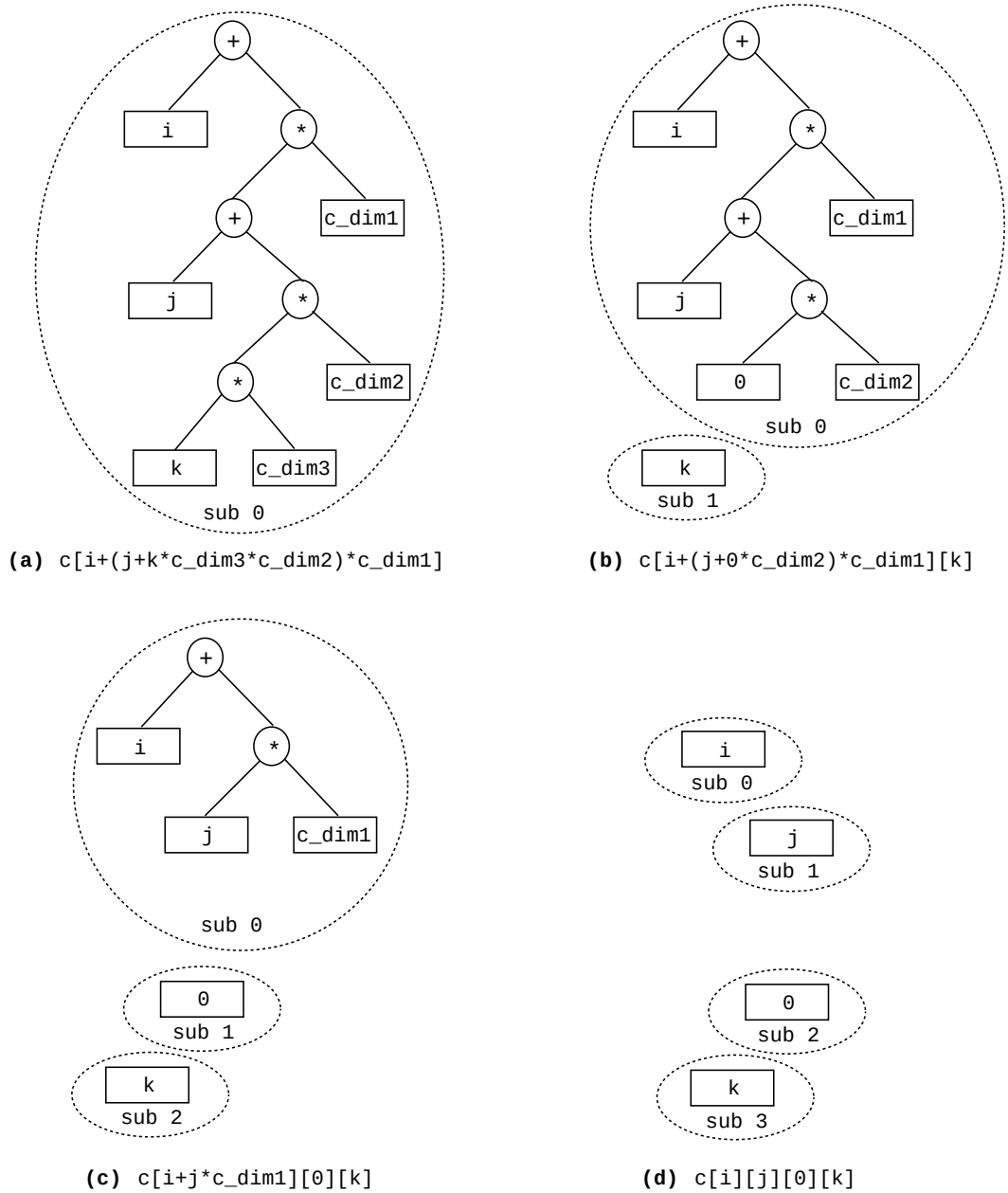
```
foo(c, n1, n2, n3, n4)
    float *c;
    int *n1, *n2, *n3, *n4;
{
    int i, j, k;
    int c_dim1, c_dim2, c_dim3, c_offset;

    c_dim1 = *n1;
    c_dim2 = *n2;
    c_dim3 = *n3 + 1;
    c_offset = c_dim1 * (c_dim2 * c_dim3 + 1) + 1;
    c -= c_offset;

    for (k = 1; k <= *n4; ++k) {
        for (j = 1; j <= *n2; ++j) {
            for (i = 1; i <= *n1; ++i) {
                c[i + (j + k * c_dim3 * c_dim2) * c_dim1] = 0.0;
            }
        }
    }
}
```

(b)

Figure 4.6: *F2c* Linearization of Formal Parameter Arrays.

by delinearization since only the relative indices between two array accesses are relevant for dependence analysis.

Delinearization of arrays that are formal parameters consists of partitioning the linearized array reference subscript expression tree into several subexpression trees based on the location of the dimension variables, such that each subexpression tree corresponds to an array subscript in the original *Fortran* code. Figure 4.7(a) shows the linearized subscript expression tree for the array reference **c** shown in Figure 4.6(b). Small circles represent binary operations while rectangles represent variables or constants in the expression tree. Each subscript expression is enclosed in a dotted ellipse for clarity. The



(a) `c[i+(j+k*c_dim3*c_dim2)*c_dim1]`

(b) `c[i+(j+0*c_dim2)*c_dim1][k]`

(c) `c[i+j*c_dim1][0][k]`

(d) `c[i][j][0][k]`

Figure 4.7: Formal Parameter Array Delinearization Example.

delinearization algorithm is divided into two phases: the search for the dimension variables in the subscript expression tree, and the partitioning of the expression tree into delinearized subexpressions.

The search for the dimension variables consists of a depth-first traversal of the nodes in the original subscript expression tree. Subtrees that represent embedded array references are not searched since they are delinearized separately. When a variable expression is encountered, the variable name is checked to see if it has a suffix of the form: `_dim<integer>`, which is due to the stylized way *f2c* generates dimension variable names. If so, a pointer to the expression tree node containing the dimension variable is saved in a dimension variable array indexed by the dimension variable number. The search phase of the algorithm is necessary to order the dimension subscripts by number for the next phase of the algorithm.

In the next phase of the delinearization algorithm, the dimension variable array is stepped through from the largest-numbered dimension variable to the smallest-numbered. Figure 4.7 illustrates the process of partitioning the subscript expression tree into delinearized subscripts (in column-major order, in order to ease comparison to the original *Fortran* code), one subscript at a time. In the first step, shown in Figure 4.7(b), the variable **k** is identified as the subscript corresponding to dimension variable **c_dim3**, since they share a common parent multiply operation. Because the parent of the multiplication expression is another multiplication expression, indicating that an explicit original subscript value is missing from the linearized subscript expression, the child multiplication

expression is replaced with the constant zero and dimension variable three is discarded. In the next step, shown in Figure 4.7(c), zero is identified as the subscript corresponding to dimension variable two. However, since the parent of the multiplication expression is now an addition expression (and the original subscript is explicit), the addition expression is replaced by its left operand, the variable **j**, and the multiplication expression is discarded along with dimension variable two. The last step, illustrated in Figure 4.7(d), proceeds in a similar manner to the previous step, and the residual expression tree becomes the left-most subscript in the delinearized array reference.

If the *Fortran* array reference contains subscripts with the lower bound index in the last dimensions (e.g. **C(I, J, 0, 0)**), the *f2c*-translated subscript expression will contain too few dimension variables.[4] Since the dimension variables for all but one of the original array dimensions are always declared in the translated code, the symbol table can be searched to check for missing higher-numbered dimension variables. Then, an extra zero subscript is added to the right side of this delinearized array reference for each dimension variable which is declared but not present in the translated array reference subscript expression.

The delinearization process just described for formal parameter arrays dimensioned by other variables whose values are not known at compile-time (called *adjustable dimensions*) is fairly simple because the original subscript expression subtrees are clearly marked by the presence of dimension variables in the linearized subscript expression. However, as

[4]If this situation occurs in any dimensions but the last, no dimension variables will be missing in the linearized subscript expression, as illustrated in Figure 4.7(a).

```
SUBROUTINE FOO(C, N2, N3)
REAL C(*)
INTEGER N2, N3

INTEGER I, J, K


DIMENSION C(10, N2, N3)




DO K = 1, N3
   DO J = 1, N2
      DO I = 1, 10
         C(I, J, K) = 0.0
      ENDDO
   ENDDO
ENDDO
RETURN
```

(a)

```
foo(c, n2, n3)
    float *c;
    int *n2, *n3;
{
    int i, j, k;
    int c_dim2, c_offset;

    c_dim2 = *n2;
    c_offset = (c_dim2 + 1) * 10 + 1;
    c -= c_offset;

    for (k = 1; k <= *n3; ++k) {
        for (j = 1; j <= *n2; ++j) {
            for (i = 1; i <= 10; ++i) {
                c[i + (j + k * c_dim2) * 10] = 0.0;
            }
        }
    }
}
```

(b)

Figure 4.8: *F2c* Linearization of Formal Parameter Arrays with Constant Dimension Size.

shown in Figure 4.8, if an array subscript other than the right-most one (for *Fortran* code) is dimensioned by a constant value, no dimension variable is declared for that subscript, and the constant value itself is instead used as the multiplier for the dimension size in the array references. The right-most array subscript dimension is not needed to calculate the linearized array reference subscript expression, and hence plays no part in array delinearization. Fortunately, if the missing variables are ignored during the second phase, the delinearization algorithm described above still applies. However, the subscript dimensioned by the constant value will remain linearized with the next subscript to the right in the resulting partially delinearized array reference. For example, the reference to array **c** in Figure 4.8 becomes the reference **c[i + j*10][k]** after delinearization. Since the subscripts for all references to the array are delinearized in a consistent manner, the

partial delinearization will result in valid data dependence analysis, although some false dependences may be calculated, as described earlier in this section. Modification of the *f2c* translator to generate dimension variables for constant dimension sizes is necessary to eliminate the false dependences calculated because of partially delinearized references.

4.3.2 Local and common block array delinearization

The delinearization of arrays local to a function and arrays contained within common blocks proceeds in a different manner than for formal function parameter arrays. Since all array dimension sizes must be constant values known at compile time, *f2c* does not generate dimension variables during the language translation process. Instead, the linearized subscript expression contains the integer constants representing the dimension sizes.

Figure 4.9 illustrates the process of *f2c* local array linearization. Note that for local and common block arrays, *f2c* preserves the original array dimensions in a comment after the array declaration. One obvious difference from the linearization of formal parameter arrays is that the offset due to differences in index lower bounds for *Fortran* and *C* is present in the linearized array subscript expression since its value is known at compile time. This offset is calculated using the same method described in Section 4.3.1. To delinearize local and common block array references, the array dimension sizes must be preserved through the *IMPACT* front-end compiler modules to the *Pcode* module, as described in Section 3.1.2.

```
SUBROUTINE FOO()

REAL C(10,20,30,40)
INTEGER I, J, K

DO K = 0, 29
   DO J = 1, 20
      DO I = 1, 10
         C(I, J, K+1, 2) = 0.0
      ENDDO
   ENDDO
ENDDO
RETURN
```

(a)

```
foo()
{
    float c[240000] /* was [10][20][30][40] */;
    int i, j, k;

    for (k = 0; k <= 29; ++k) {
        for (j = 1; j <= 20; ++j) {
            for (i = 1; i <= 10; ++i) {
                c[i + (j + (k + 61) * 20) * 10 - 6211] = 0.0;
            }
        }
    }
}
```

(b)

Figure 4.9: *F2c* Linearization of Local Arrays.

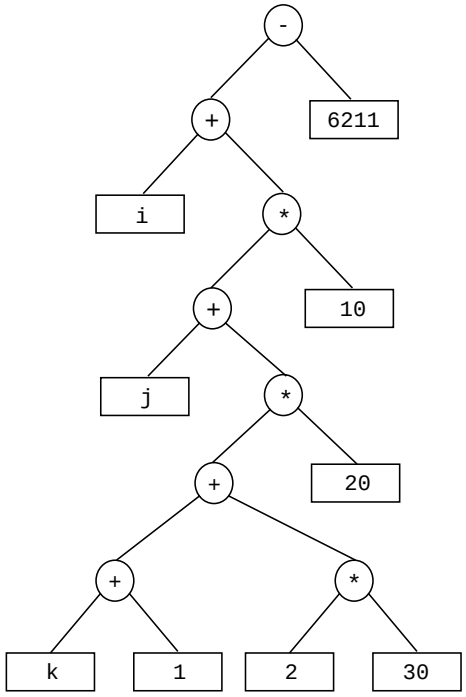
In *f2c*-translated code, a dimension constant may be folded with other dimension constants, with other constants from the original subscript expressions (as shown in Figure 4.9 for dimension constant 30 and subscript expression constants one and two), or with the offset constant. Since constant folding makes the process of delinearization much more complicated if not impossible in some cases, the *f2c* translator was modified to prevent the folding of relevant constants within array subscript expressions. If constant folding is suppressed during *f2c* translation, the array reference shown in Figure 4.9(b) becomes

$$\mathbf{c[i + (j + ((k + 1) + 2 * 30) * 20) * 10 - 6211]}$$

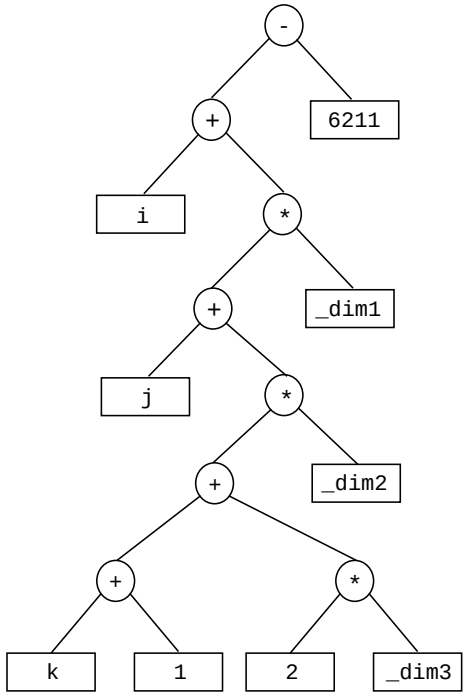
Note that all useful dimension constants and subscript constants are now represented explicitly in the linearized subscript expression.

This delinearization process is more complicated than for delinearization of formal parameters, as described in Section 4.3.1, because the dimension constants in the linearized subscript expression must be distinguished from one another and from other integer constants present in the original subscript expressions. However, if the dimension constants are identified and replaced by the appropriate dummy dimension variables, the delinearization algorithm for formal parameter arrays may be applied directly. Figure 4.10(a) illustrates the linearized array subscript expression tree for the array reference **c** given above (with constant folding suppressed), and Figure 4.10(b) shows the linearized subscript expression after replacement with dummy dimension variables. The algorithm to identify and convert the dimension constants to dummy dimension variables is divided into three phases: gathering the potential dimension paths, selection of the path which matches the known dimension constants, and replacement of the selected dimension constants with corresponding dummy dimension variables.

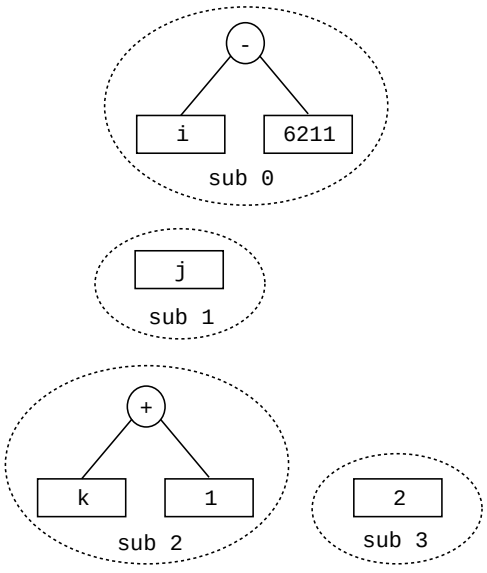
In the first phase of the algorithm, all potential dimension paths are enumerated in order to simplify determination of the actual dimension constants. A *potential dimension constant* is defined as any integer constant which is a child of a multiplication operator in the subscript expression tree. No other integer constants are considered potential dimension constants since suppression of constant folding prevents other operators from being parents of true dimension constants. For example, **10**, **20**, **2**, and **30** are the potential dimension constants present in the subscript expression tree shown in Figure 4.10(a). A *potential dimension path* is defined as an ordered list of potential dimension constants



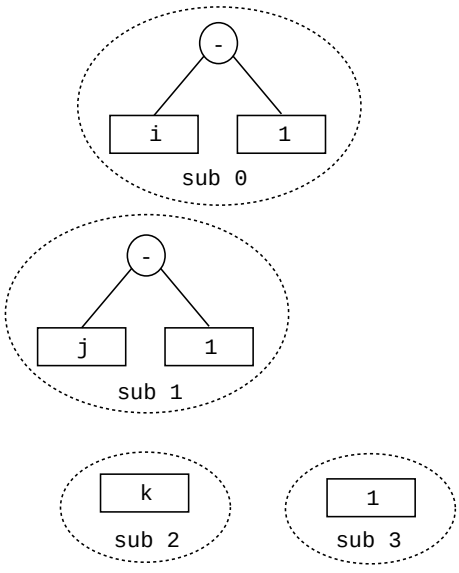
(a) `c[i+(j+((k+1)+2*30)*20)*10-6211]`



(b) `c[i+(j+((k+1)+2*_dim3)*_dim2)*_dim1-6211]`



(c) `c[i-6211][j][k+1][2]`



(d) `c[i-1][j-1][k][1]`

Figure 4.10: Local Array Delinearization Example.

encountered as children of multiplication operators along a path from a potential dimension constant to the root of the subscript expression tree. The set of potential dimension paths in the example subscript expression tree shown in Figure 4.10(a) is as follows:

$$\{(\mathbf{10}), (\mathbf{20}, \mathbf{10}), (\mathbf{2}, \mathbf{20}, \mathbf{10}), (\mathbf{30}, \mathbf{20}, \mathbf{10})\}$$

The procedure to determine the potential dimension paths consists of a depth-first traversal of the subscript tree and a walk to the root of the subscript expression tree each time a potential dimension constant is encountered to record the potential dimension path beginning with that dimension constant. If the number of potential dimension constants in a potential dimension path is not equal to the number of array dimensions minus one, the path is discarded. Consequently, the set of potential dimension paths found by the algorithm for the example subscript expression tree in Figure 4.10(a) is as follows:

$$\{(\mathbf{2}, \mathbf{20}, \mathbf{10}), (\mathbf{30}, \mathbf{20}, \mathbf{10})\}$$

The second phase of conversion to dummy dimension variables consists of the identification of the potential dimension path containing the actual dimension constants. First, the actual dimension sizes for all but the right-most array dimension are extracted from the delinearization pragma (see Section 3.1.2) found in the symbol table entry for the array variable. Next, the potential dimension constants for each potential dimension path are matched to the extracted array dimension sizes in reverse order.

Since constant folding is suppressed in the *f2c* translator, at least one potential dimension path must match the extracted dimension sizes completely. For the example in

Figure 4.10(a), the matching potential dimension path is $(\mathbf{30}, \mathbf{20}, \mathbf{10})$ since the original array dimensions are $\mathbf{c}[\mathbf{10}][\mathbf{20}][\mathbf{30}][\mathbf{40}]$. If there is more than one matching potential dimension path, then one of the potential dimension paths is chosen arbitrarily. This situation arises, for example, if the array reference in Figure 4.9(a) had instead been $\mathbf{C}(\mathbf{I}, \mathbf{J}, \mathbf{K} + \mathbf{1}, \mathbf{30})$, such that the right-most subscript consists of an integer constant with the same magnitude as the size of the second-to-last array dimension.

The third phase of the conversion algorithm consists of the actual conversion of the dimension constants along the chosen dimension path to the corresponding dummy dimension variables as shown in Figure 4.10(b). The dummy dimension variables are temporary variables used for convenience; they allow the delinearization algorithm for formal parameter arrays to be applied directly to the converted subscript expression. Figure 4.10(c) shows the results of applying this delinearization algorithm to the subscript expression tree shown in Figure 4.10(b).

Since array delinearization is applied consistently and since the offset constant is the same for all references to the same array, the array reference may at this point be considered delinearized for the purposes of data dependence analysis. In fact, a constant offset applied consistently to the same subscript of all references to a given array does not affect data dependence analysis results. However, the linearized constant offset (**6211** in Figure 4.10(b)) calculated from the difference in lower bounds using all subscripts is now located entirely in the leftmost subscript.

Because of the difficulty in associating these subscript expressions with those in the original *Fortran* code while examining access table listings, the delinearized subscripts are each adjusted by compensating offsets, such that each resulting subscript is individually offset with the *Fortran* default lower bound of one. The linearized offset, ω, introduced by *f2c* is calculated for an array dimensioned in *Fortran* as $A(d_1, d_2, \ldots, d_n)$ via the following formula:

$$\omega = 1 + \sum_{i=1}^{n-1} \prod_{j=1}^{i} d_j$$

This linearized offset, 6211 for the example in Figure 4.9, is added to the left-most subscript expression, and the value of one is subtracted from each subscript in the delinearized reference. Figure 4.10(d) shows the resulting array reference after this subscript adjustment process is performed (and constant folding is applied to the resulting expressions). Note that each subscript index is now one less than in the original code shown in Figure 4.9(a).

Unfortunately, if an array is dimensioned in the *Fortran* source with a subscript index lower bound other than one, the adjustment process described above will not result in each subscript having the correct delinearized offset. Since *f2c*-converted code contains no information about the subscript index lower bounds, exact delinearization of the linearized offset is not possible without extensive modification of the *f2c* translator. However, as mentioned above, data dependence analysis is not affected by this adjustment process, and should still produce correct results.

Although correct and complete delinearization for data dependence analysis is realizable using the algorithms described above when the original array dimension sizes are known, arrays for which dimension size information is unavailable after *f2c* translation cannot be delinearized using these algorithms. For example, *f2c* does not generate dimension size information for *equivalenced* arrays which are offset in memory, described in Section 4.2.4. Unfortunately, multidimensional offset *equivalenced* arrays do occur in real programs, and the *f2c* translator requires extensive modification to generate dimension size information in order to delinearize these references. However, the current delinearization algorithms are successful in delinearizing the vast majority of multidimensional array references.

4.4 Array Reference Subscript Analysis

After array reference delinearization, each array subscript index expression is transformed to a canonical representation to simplify and enable further analyses and transformations. Data dependence analyses typically require that array subscript expressions be represented in terms of a linear (or *affine*) function of the enclosing loop index variables and symbolic constants. Furthermore, certain array optimizations require that array references be expressed in affine form to be considered for transformation. Section 4.4.1 discusses the conversion of array subscript index expressions to a canonical

affine representation, and Section 4.4.2 presents the method, for the purpose of data dependence analysis, to distinguish symbolic constants from variables which are modified in the function.

4.4.1 Conversion to affine representation

The purpose of the first phase of array reference subscript analysis is to convert each subscript index expression into a canonical affine representation, if possible, described by the following arithmetic expression:

$$c_0 + c_1 v_1 + c_2 v_2 + \cdots + c_n v_n,$$

such that the c symbols represent integer constants or coefficients and the v symbols represent scalar program variables. Although algebraic simplification of the subscript expression would yield the desired arithmetic expression, a more direct approach is employed to convert a subscript expression.

Figure 4.11 presents the algorithm AFFINE_EXPRESSION which takes an expression tree, T, as an argument and returns either an *affine expression*, if a canonical affine representation exists for T, or a pre-defined constant, NOT_AFFINE, otherwise. An *affine expression* is a data structure that consists of a field for an array of *affine terms*, *term*[ ], and a field specifying the number of affine terms, *num_terms*, present in the affine expression. An *affine term* is itself a data structure consisting of an integer coefficient field, *coef*, and a field which is a pointer to the variable access in the access table,

```
AFFINE_EXPRESSION(T)
1    A ← ALLOCATE(affine_expr)
2    A.num_terms ← 1
3    A.term[0].var_access ← NIL
4    if LINEARITY(A,T) ≤ 1          /* Constant or Linear Expression */
5       k ← COEFFICIENT(T, NIL)
6       A.term[0].coef ← k
7       for i ← 1 to A.num_terms
8           A.term[i].coef ← COEFFICIENT(T, A.term[i].var_access) − k
9           tag(var_entry(A.term[i].var_access)) ← NOT_FOUND
10      return A
11   else
12      for i ← 1 to A.num_terms
13          tag(var_entry(A.term[i].var_access)) ← NOT_FOUND
14      FREE(A)
15      return NOT_AFFINE
```

Figure 4.11: Algorithm for Building an Affine Expression.

var_access. Thus, the affine expression data structure embodies all of the characteristics of the canonical affine representation, as given above.

The first three lines of algorithm AFFINE_EXPRESSION perform the allocation of an affine expression data structure, initialize *num_terms*, and set the constant term *var_access* pointer to NIL since the constant term, *term*[0], has no associated variable. If the expression tree is a linear function, as determined by algorithm LINEARITY, the coefficients are determined in lines 5-8 via the algorithm COEFFICIENT, and the affine expression is returned. However, if the expression tree is not a linear function, then the affine expression is deallocated and the value NOT_AFFINE is returned. The details of

```
LINEARITY(A, T)
1    case opcode(T) of
2         INTEGER:
3              return 0        /* Constant Expression */
4         VARIABLE:
5              v ← var_access(T)
6              if tag(var_entry(v)) = NOT_FOUND
7                   tag(var_entry(v)) ← FOUND
8                   A.term[A.num_terms].var_access ← v
9                   A.num_terms ← A.num_terms + 1
10             return 1        /* Linear Expression */
11        NEGATE:
12             return LINEARITY(A, first_oper(T))
13        ADD:
14             return MAX(LINEARITY(A, first_oper(T)), LINEARITY(A, second_oper(T)))
15        SUBTRACT:
16             return MAX(LINEARITY(A, first_oper(T)), LINEARITY(A, second_oper(T)))
17        MULTIPLY:
18             return LINEARITY(A, first_oper(T)) + LINEARITY(A, second_oper(T))
19        DEFAULT:
20             return 2        /* Non-linear Expression */
```

Figure 4.12: Algorithm for Determining Whether an Expression Is Linear.

the two algorithms, LINEARITY and COEFFICIENT, called from AFFINE_EXPRESSION are explained below to further the understanding of the creation of the affine expression.

Figure 4.12 shows the algorithm LINEARITY, which takes an affine expression, A, and an expression tree, T, as arguments, and returns whether or not the expression tree can be represented using an affine expression data structure. The algorithm performs two logical functions while traversing the entire expression tree once.

First, LINEARITY determines the number of terms in the affine expression by counting the unique program variables accessed within the expression tree. Each time a variable

reference is found in the expression tree (line 4 of Figure 4.12), the corresponding variable access is located in the access table. Each variable entry in the access table contains a field, *tag*, indicating whether or not an instance of that particular variable has already been found in the expression tree traversal. Each time a new variable reference is encountered (line 6), the variable entry is tagged FOUND (line 7), and the number of terms in the potential affine expression is increased by one (line 9). Additionally, a pointer to the variable access in the access table is stored in the appropriate affine term data structure to identify the variable corresponding to that term (line 8). Note that the variable entry tags which may be marked FOUND by LINEARITY are reset to NOT_FOUND in lines 9 and 13 of algorithm AFFINE_EXPRESSION (Figure 4.11).

The second logical function of LINEARITY is to determine whether the expression tree describes a linear function of scalar variables. To simplify the determination, however, the return value of LINEARITY was chosen as the degree of the polynomial described by the expression tree. Therefore, an integer constant node is of degree zero (line 3), a variable node is of degree one (line 10), and any inherently non-linear expression tree node, i.e., a function call operator or array index operator, is assigned degree two (line 20), implying non-linearity. To calculate the polynomial degree for the whole expression, the leaf node polynomial degrees are combined through the parent arithmetic operators: negation, addition, subtraction, and multiplication. For an addition/subtraction operator, the polynomial degree of the add/subtract expression is the maximum of the polynomial degrees of the operator's operand expressions (lines 14 and 16). However, for

```
COEFFICIENT(T, v)
1    case opcode(T) of
2         INTEGER:
3              return value(T)
4         VARIABLE:
5              if var_entry(v) = var_entry(var_access(T))
6                return 1
7              else
8                return 0
9         NEGATE:
10             return −COEFFICIENT(first_oper(T), v)
11        ADD:
12             return COEFFICIENT(first_oper(T), v) + COEFFICIENT(second_oper(T), v)
13        SUBTRACT:
14             return COEFFICIENT(first_oper(T), v) − COEFFICIENT(second_oper(T), v)
15        MULTIPLY:
16             return COEFFICIENT(first_oper(T), v) ∗ COEFFICIENT(second_oper(T), v)
```

Figure 4.13: Algorithm for Finding the Coefficient of an Affine Term.

a multiplication operator, the resulting polynomial degree is the sum of the polynomial degrees of the operand expressions (line 18).

If the LINEARITY algorithm determines that the expression tree is a constant or linear function of the variables (line 4 of Figure 4.11), then the coefficient of the variable stored in each affine term is calculated from the expression tree by the algorithm COEFFICIENT given in Figure 4.13. COEFFICIENT takes as arguments an expression tree, T, and an optional pointer, v, to a variable access in the access table, and works by evaluating the expression tree with the variables set to either the value one or the value zero. If all variable values are set to zero by passing NIL for v, as in line 5 of AFFINE_EXPRESSION, the result of evaluating the expression tree is the value of the constant term in the affine

expression. However, if v is not NIL, all instances of the variable v evaluate to one, and all other variables evaluate to zero as shown in lines 5-8 of the algorithm COEFFICIENT. In this case, the value returned by COEFFICIENT is the constant term value plus the coefficient of the variable v. Therefore, the constant term value, k, is subtracted from the coefficient for v before the coefficient is stored in the appropriate affine term data structure, as shown in line 8 of the algorithm AFFINE_EXPRESSION.

Note in lines 7 and 8 of algorithm AFFINE_EXPRESSION that COEFFICIENT is called once for each unique variable in the expression tree. Thus, the expression tree is traversed as many times as there are terms in the resulting affine expression. Although the run-time complexity of the algorithms presented in this section seems to be greater than that of an equivalent algebraic manipulation implementation, the complexity may not be the determining factor for the execution time, since the individual subscript expressions tend to have a small number of terms. Furthermore, the constant cost of algebraic manipulation may be larger because of the allocation and freeing of expression nodes required during the simplification process.

4.4.2 Identification of modified variables

Although the array subscript expressions are converted to an affine representation, this representation includes not only loop index variables and symbolic constants but also variables which may not have constant values during some part of the function execution, called *modified variables*. The presence of these modified variables may prohibit

correct dependence analysis if they are not identified, since the *Pcode* data dependence analysis phase assumes all non-index variables are symbolic constants. Furthermore, array optimizations and transformations may require that symbolic constants and modified variables be distinguished.

For the purpose of analysis and optimization, the same variable may be identified as a symbolic constant within a specified lexical region of a source function and a modified variable within another region. For modified variable identification, the *IMPACT* data dependence analyzer considers each loop nest in a function as a separate region, and the remaining code which is not enclosed within a loop is also considered one region, called the *top-level region*. Section 5.4.2 presents the motivation for this choice of region for data dependence analysis.

The first step of modified variable identification is to construct, for each variable, a set of regions within which that variable is modified. Each variable entry data structure in the access table contains a field which identifies the set of regions for which that variable is modified. The loop nest regions are numbered in lexical order, and these numbers are used to identify the regions in a region set. During access table generation, whenever a write access is added to the access table, the number of the region within which the access occurs is added to the region set for the variable written.

In the second step of modified variable identification, region sets are merged for aliased variables. An alias between two variables implies that a write to one variable is a potential write to the aliased variable. Therefore, during alias graph construction, the

region sets for the two aliased variables are combined using the union operation as the alias edge is added to the alias graph.

At this point, all variables in the access table are identified by region as either symbolic constants or (potentially) modified variables. However, for all array access subscripts, data dependence analysis needs to know if the subscript contains at least one modified variable. Therefore, after conversion of each subscript expression to affine representation, each subscript data structure in the variable access data structure is marked with a value indicating whether the subscript contains variables modified in the region containing the array access, and whether the subscript contains variables modified in the top-level region. (Loop index variables for loops which enclose the array reference are excluded from consideration as modified variables.) How this information is used to guide data dependence analysis is discussed in Section 5.4.2.

5. DATA DEPENDENCE ANALYSIS

After the variable reference analysis has been performed, the data dependence analysis phase is invoked to gather information about the loops in the program and calculate the data dependence information. The Omega Test [2], developed by William Pugh at the University of Maryland, is employed to generate and solve the data dependence equations and inequalities in order to find the distance and direction vectors for a pair of variable references. The data dependence analysis phase gathers the remaining necessary information and applies the Omega Test to appropriate pairs of references in the variable access table. Virtually all of the information calculated in the variable reference analysis phase is utilized by the data dependence analysis phase, along with loop information and information about the execution order of variable references determined in the data dependence analysis phase itself. The sections of this chapter describe, respectively, the Omega Test, loop preparation, determination of the execution order of variable references, and the generation of the data dependence information.

5.1 The Omega Data Dependence Test

The Omega Test is an integer programming algorithm tailored to solve data dependence analysis problems efficiently. It is based on an extension of Fourier-Motzkin variable elimination, a linear programming method, to integer programming. The Omega Test determines whether an integer solution exists to a set of linear equations and inequalities, generated by the three constraints for data dependence given in Section 2.3.1.

If no solution exists, the Omega Test reports that no data dependence exists; otherwise, the set of linear equations and inequalities is *symbolically projected* onto a set of new variables which represent the elements of the dependence distance/direction vector. Symbolic projection eliminates designated variables from an integer programming problem and creates a set of projected problems in terms of the remaining variables, such that the projected problems have the same integer solutions as the original problem [2]. By selective application of symbol projection, the elements of the direction vector can be determined much more efficiently than deciding whether dependences exist for each possible direction vector separately.

Typically, data dependence tests have relied on *loop normalization*, which converts a loop into one with a constant lower bound and unit step, to simplify data dependence testing. However, loop normalization can result in data dependences with non-integral dependence distances, which often go undetected using traditional data dependence testing [11]. The Omega Test uses the definition of dependence distance which uses the difference in the values of the index variables (sometimes called *dependence difference*),

so that loop normalization need not be applied. However, this definition of dependence distance is problematic for loops with negative step, since distance/direction vectors are not guaranteed to be lexicographically positive [2]. Indeed, the validity of several common loop transformations cannot be determined without the assumption that distance/direction vectors are always lexicographically positive for real data dependences. Therefore, the Omega Test currently may be applied only to loops with constant positive step. (Loops with step which cannot be exactly determined at compile time are excluded since the step at run time may be negative.) Although this restriction diminishes the applicability of the Omega Test to certain loops, no dependences remain undetected due to nonintegral dependence distance.

The Omega Test also performs value-based dependence analysis [9], [10] for code which contains only **if** and **for** *C* language control flow constructs; **break**, **continue**, **return** and **goto** constructs are not handled directly by the Omega Test interface. Although application of the test to code containing the excluded constructs is possible in certain cases, applying version 2.0.1 of the Omega Test to arbitrarily complex patterns of these control constructs is currently infeasible.

Version 2.0.1 of the interface to the Omega Test for data dependence analysis consists of three parts:

1. Functions which access information about variable references and loops.

2. Data structures which transfer information to and from the Omega Test.

3. Driver code which applies the Omega Test to pairs of variable references.

The functions which access information are often represented by *C* language preprocessor macros which directly access the variable reference information gathered by the reference analysis phase described in Chapter 4. Although the Omega Test Interface provides samples of such functions, the functions were adapted to work with the *Pcode IR*. The affine representation for array subscripts, loop bound information, and dependence distance and direction vectors are examples of data structures which communicate information between the Omega Test and the data dependence analyzer. The Omega Test driver code must be adapted to the particular form of variable access data structure employed by the dependence analyzer. The driver code adaptation for the *Pcode* data dependence analyzer is discussed in more detail in Section 5.4.

5.2 Loop Preparation

Loop-carried data dependence analysis requires information about loops which enclose the variable references to be analyzed. Specifically, this loop preparation process consists of three components:

1. Standardization of loops to a format suitable for data dependence analysis.

2. Determination of nesting relationships among standardized loops.

3. Analysis of loop bound expressions.

The three sections which follow describe each of these components, respectively.

5.2.1 Loop standardization

Loop-carried data dependence analysis typically places certain restrictions on the properties of loops recognized by the analysis. For example, many data dependence analyzers require that loop index variables are increased or decreased by a constant step size at the end of each iteration, which is enforced by the *Fortran* **do** loop semantics. However, the *Fortran* **do while** loop does not always contain a recognizable index variable, rendering such a loop unrecognizable by many dependence analyzers. The *loop standardization* procedure standardizes certain loops in the *f2c*-generated source code to a type of loop called a *Parloop*, which has the same semantics as the *Fortran*-style **do** loop.

The *Pcode* data dependence analyzer employs three criteria for loop standardization:

1. The loop must originate from a *Fortran* **do** loop.

2. The loop must have a positive step size known at compile time.

3. The loop must not be nested more deeply than six levels.

The first criterion insures that the loop has an identifiable index variable. Although some while loops may also have identifiable index variables, the current implementation of the data dependence analyzer does not include this capability.

The second criterion insures that the distance/direction vector representation is lexicographically positive, as described above in the introduction to Section 5.1. This criterion could be circumvented by converting loops with negative step to loops with positive

step of the same magnitude — a process similar to loop normalization, but without the problems described above. Since loops with negative step occur infrequently in practice, this circumvention has not yet been implemented in the data dependence analyzer.

The third criterion is a result of the restrictive data structure that the Omega Test uses to represent direction vectors. The data structure is represented as a bit vector which must fit in a 32-bit integer. Because many of the bits are reserved for value-based dependence analysis flags, only six 3-bit direction vectors are supported. This restriction can be relaxed to 10 levels of loop nesting by redefining the bit vector fields, or even more levels by redefining the data structure for direction vectors. However, the later requires modifications to nearly all of the Omega Test source code.

Loops which meet all three criteria can be handled by the dependence analyzer and are standardized to *Parloop* format. During this standardization process, the initial, final, and increment values of the index variable are identified from the *C* language **for** loop header expressions and stored in fields of the *Parloop* data structure, which is a statement node of the *Pcode IR*. In addition, the index variable itself is stored in a field of the *Parloop* data structure for easy identification.

Loops which fail to meet any of the three standardization criteria are not recognized as loops for the purpose of loop-carried data dependence analysis. Data dependence analysis may still be applied to loop nests containing non-standardized loops, but the data dependence direction and distance vectors will contain a missing entry for each non-standardized loop. Many program transformations and optimizations must take into

account the missing vector entries when testing for validity, which may unnecessarily prevent these transformations and optimizations. Furthermore, any non-standardized loop index/induction variable appearing in an array subscript is treated as a modified variable, which results in even more conservative data dependence analysis. Expanding the number of standardizable loops is an area of ongoing research.

5.2.2 Loop nesting determination

Data dependence analysis for a pair of variable references requires access to loop bound and step information for all loops which enclose the reference pair. Hence, each standardized loop is linked via a pointer to its nearest-enclosing, or parent, standardized loop. The parent standardized loop is located by following statement parent pointers upward through the *AST* until a standardized loop is encountered. Furthermore, the nesting depth of each standardized loop is annotated to the *Parloop* data structure before dependence analysis begins. Non-standardized loops are skipped when generating these parent pointers and depth annotations since they have no corresponding entries in the data dependence distance/direction vectors. The standardized loop parent pointers and depth annotations directly provide the necessary loop nesting information to the Omega Test interface.

5.2.3 Loop bound analysis

The Omega Test requires that each loop bound expression be a linear combination of the outer enclosing loop index variables and symbolic constants in order to generate

index variable constraints for the data dependence analysis problem. Hence, upper and lower bounds of standardized loops are converted to an affine representation as described previously in Section 4.4.1.

The Omega Test also accepts certain cases of **min** and **max** functions as part of the affine representation for loop bounds. One **max** function may be present in a positive arithmetic term or one **min** function in a negative arithmetic term for the loop lower bound. Conversely, one **min** function may be present in a positive term or one **max** function in a negative term for the loop upper bound. If these conditions do not hold, the loop bound is marked non-affine and the corresponding constraint on the index variable is not generated by the Omega Test.

A loop bound which contains a **min** or **max** function accepted by the Omega Test as an affine representation generates the conjunction of two separate constraints for the loop index variable. For instance, for a loop with upper bound, **min**(a, b), and index variable, i, the constraints generated for constraint two of Section 2.3.1 are as follows:

1. $i \leq a$

2. $i \leq b$

The restrictions for **min** and **max** functions to be handled only in certain contexts are necessary since the Omega Test solves the conjunction of several constraints — disjunctions of constraints are not handled.

The representation for an affine expression containing a **min** or **max** function is just two affine expressions: one representing the affine expression if the result of the **min**

or **max** function is the first argument of the function, and one representing the affine expression if the result is the second argument. Modifications for the affine representation algorithms to incorporate handling of **min** and **max** functions are straightforward and are not presented in this thesis.

Modified variables which appear in loop bounds are detected during variable reference analysis according to the process described in Section 4.4.2. Any modified variable appearing in a loop bound causes the data dependence analyzer to mark the bound as non-affine, thereby preventing the Omega Test from generating the corresponding index variable constraint for that bound. Although overly conservative data dependences may be generated from using this approximation, it prevents the data dependence analyzer from treating the modified variable as a symbolic constant. Treating modified variables as symbolic constants could cause some data dependences to have incorrect direction/distance vector information, and could cause other data dependences to go undetected.

5.3 Reference Ordering Determination

For loop-carried data dependence analysis, the mathematical formulation of constraint one given in Section 2.3.1 completely determines the temporal ordering between the two references undergoing analysis, since the two references are from different iterations of at least one of the common enclosing loops. However, the iteration vectors are identical for the non-loop-carried data dependence case (the case for which the distance vector

is all zeros), which necessitates the use of control flow and expression tree analyses to determine the execution ordering of references. The next two sections describe each of these analyses in detail.

5.3.1 Non-loop-carried reachable control flow analysis

Since the control flow graph is constructed at the granularity of the statement level of the *Pcode AST*, control flow information is sufficient to determine the reference ordering information between references in different expression trees. To determine whether reference **A** can precede reference **B** during the execution of a single iteration of their innermost enclosing loop, **L**, we need only determine whether there is a control flow path from **A** to **B** which does not include the loop back-branch of **L**. If so, the dependence analyzer must consider the possibility that reference **B** is data dependent on reference **A**. If not, the reference **B** cannot be data dependent on reference **A** in the non-loop-carried sense. As an example of the latter case, two references in different branches of an **if** statement have no control flow path between them, so there is no need to consider the possibility of non-loop-carried dependence.

To determine whether or not there is a control flow path from one point in a function to another, control flow analysis is employed to calculate all the reachable control flow nodes at each control flow node in the function. However, since it is trivial to determine reachability within each basic block, the reachable control flow analysis is performed at the granularity of basic blocks. In addition, since the criteria for determining reference

ordering are with respect to the innermost loop (**L**) enclosing both references, the reachability information for each basic block must be calculated with respect to each enclosing standardized loop as well as the entire function (for the case where there are no common enclosing loops for the reference pair). Since standardized loops can be nested only six levels deep, each basic block contains an array of six sets of reaching basic block identifiers, such that the array index value is the depth of the standardized loop for which each set corresponds.

Figure 5.1 provides the algorithm for calculating the reaching basic block sets. The algorithm is formulated similar to a forward data flow analysis, and takes a *Pcode* function data structure as an argument. Line 10 of Figure 5.1 shows that the reaching basic blocks for the current basic block are the immediate predecessor basic blocks in the current *Parloop* plus the union of all of their reaching basic blocks. Before the algorithm is invoked, all reaching basic block sets are initialized to the empty set. The reaching basic block sets are calculated repeatedly until there are no more changes to any set. Note that line 7 of the algorithm prevents reaching basic blocks from traversing the loop back-branch for the current *Parloop* by inhibiting the data flow calculation for the loop header basic block.

Lines 14–23 of REACHING_BASIC_BLOCKS calculate the reaching basic blocks for the entire function. The differences from the previous calculation are that no loop back-branches are ignored and the loop nesting depth is set at zero.

```
REACHING_BASIC_BLOCKS(F)
1   for each P ∈ parloops(F)
2       D ← loop_nesting_depth(P)
3       Change ← TRUE
4       while Change = TRUE
5           Change ← FALSE
6           for each B ∈ basic_blocks(P)
7               if ¬ is_header_bb_for_loop(B, P)
8                   R ← ∅
9                   for each B' ∈ predecessor(B) ⋂ basic_blocks(P)
10                      R ← R ⋃ {B'} ⋃ reaching_bb's(B', D)
11                  if R ≠ reaching_bb's(B, D)
12                      reaching_bb's(B, D) ← R
13                      Change ← TRUE
14  Change ← TRUE
15  while Change = TRUE
16      Change ← FALSE
17      for each B ∈ basic_blocks(F)
18          R ← ∅
19          for each B' ∈ predecessor(B)
20              R ← R ⋃ {B'} ⋃ reaching_bb's(B', 0)
21          if R ≠ reaching_bb's(B, 0)
22              reaching_bb's(B, 0) ← R
23              Change ← TRUE
```

Figure 5.1: Algorithm for Calculating Reaching Basic Blocks.

Although the algorithm could be modified to calculate the information for all loop nests simultaneously, preventing the reaching basic blocks from propagating outside of their enclosing *Parloops* adds computational complexity to the algorithm. If the unnecessary propagation of reaching basic blocks is not prevented, the sets become unnecessarily large, resulting in slower set operations and wasted storage space.

To determine if reference **A** may precede reference **B** within the execution of the same iteration of their innermost enclosing *Parloop*, **L**, the data dependence analyzer

first checks whether or not **A** and **B** are in the different basic blocks of **L**. If so, **A** may precede **B** if the basic block containing **A** is present in the set of reaching basic blocks at the nesting depth of **L** for the basic block containing **B**. If **A** and **B** are in the same basic block, but different flow nodes within that basic block, a simple traversal of the flow-node list will determine the possible ordering of reference for **A** and **B**. However, if **A** and **B** are within the same control flow node, then the expression corresponding to that flow node must be analyzed to determine the reference ordering, as discussed in the next section.

5.3.2 Intra-expression execution order analysis

If two references, **A** and **B**, are contained within the same expression AST, a method for determining the execution ordering of **A** and **B** other than control flow analysis must be employed to determine whether non-loop-carried data dependences can exist from one reference to the other. The execution ordering for two references can be determined by examining the type of operation which constitutes their nearest common ancestor in the expression tree.

Table 5.1 specifies the orders of evaluation for the operand expressions of binary and ternary C language operations. The relation, "$\prec$", specifies that evaluation of the expression on the left side occurs before evaluation of the right-side expression. Hence, references in the left-side expression will execute before references in the right-side expression. The relation, "?", specifies that the order of evaluation of the operands is not

Table 5.1: Orders of Evaluation for Expression Operands.

Operation Type	Example Expression	Order of Evaluation
Assignment [1]	$\alpha = \beta$	$\beta \prec \alpha$
Logical	$\alpha \;\&\&\; \beta$	$\alpha \prec \beta$
Sequential [2]	$\alpha,\ \beta$	$\alpha \prec \beta$
Other Binary [3]	$\alpha + \beta$	$\alpha \;?\; \beta$
Conditional	$\alpha \;?\; \beta : \gamma$	$\alpha \prec \beta$ $\alpha \prec \gamma$ $\beta \not\prec \gamma \ \wedge\ \gamma \not\prec \beta$

mandated by the *C* language; therefore, no non-loop-carried dependence is generated between the references. Similarly, no non-loop-carried dependence is generated between references contained in the second and third operands of the conditional operator, since only one of the operands is evaluated each time the conditional expression is evaluated.

5.4 Data Dependence Graph Generation

The final step of data dependence analysis is to generate and record the data dependence information for pairs of variable references. The data structure chosen to represent the dependence information is a directed graph, such that the nodes represent program references and arcs represent data dependences between them. To generate this information, one must first iterate through all pairs of possibly data-dependent references in the

[1] For the compound assignment operators, (+=, −=, ∗=, /=, %=, <<=, >>=, &=, ^=, and |=), the left-hand-side write reference (in α) must execute after any of the references on the right-hand side (in β); however, the left-hand-side read reference may execute either before or after references on the right-hand side.

[2] Argument expressions for a function call invocation are not necessarily sequentially evaluated. References within different argument expressions may execute in any order.

[3] This order of evaluation applies to any binary operation in the *C* programming language whose specific operation type does not appear in Table 5.1.

access table. This iterative process of "intersecting" each program reference with other program references is described in Section 5.4.1. Preparation of each pair of references for application to the Omega Test is accomplished by the Omega Test driver, which is described in Section 5.4.2. Finally, a more detailed description of the data dependence graph representation is given in Section 5.4.3.

5.4.1 Pairwise reference intersection

Figure 5.2 contains the algorithm for reference intersection, which takes the access table as an argument. The subroutine TEST_DRIVER performs the final preparation and calls the Omega Test for the variable accesses specified by the first and second arguments. The third argument is the data dependence type for $A1\ \delta\ A2$, and the fourth argument is the type for $A2\ \delta\ A1$. (Input-type data dependences are not typically calculated.) Macros *subscript_list*, *last_subscript*, and *next_subscript* are utilized to manipulate data structures represented as linked lists. Lines 1–10 intersect references from the same variable entry and the same variable type data structures. Since the subscript expressions for these accesses are directly comparable, subscript expression lists remain attached to the pair of accesses and, thus, are passed to the Omega Test to generate constraints.

Although pairwise reference intersection seems to be straightforward, aliased variables complicate the process significantly. Recall from Section 4.1.2 that references to variables of different types which overlap in memory are organized by data type in the access table. Those array references which are placed under different variable type data structures but

```
INTERSECT_REFERENCES(AT)
1     for E1 ← first_entry(AT) to last_entry(AT)
2           for T1 ← first_type(E1) to last_type(E1)
3                 for A1 ← first_access(T1) to last_access(T1)
4                       for A2 ← first_access(T1) to A1
5                             if is_read(A1) ∧ is_write(A2) ∧ A1 ≠ A2
6                                   TEST_DRIVER(A1, A2, ANTI, FLOW)
7                             if is_write(A1) ∧ is_read(A2)
8                                   TEST_DRIVER(A1, A2, FLOW, ANTI)
9                             if is_write(A1) ∧ is_write(A2)
10                                  TEST_DRIVER(A1, A2, OUTPUT, OUTPUT)

11                for T2 ← next_type(T1) to last_type(E1)
12                      for A1 ← first_access(T1) to last_access(T1)
13                            S1 ← subscript_list(A1)
14                            subscript_list(A1) ← NIL
15                            for A2 ← first_access(T2) to last_access(T2)
16                                  S2 ← subscript_list(A2)
17                                  subscript_list(A2) ← NIL
18                                  if is_read(A1) ∧ is_write(A2)
19                                        TEST_DRIVER(A1, A2, ANTI, FLOW)
20                                  if is_write(A1) ∧ is_read(A2)
21                                        TEST_DRIVER(A1, A2, FLOW, ANTI)
22                                  if is_write(A1) ∧ is_write(A2)
23                                        TEST_DRIVER(A1, A2, OUTPUT, OUTPUT)
24                                  subscript_list(A2) ← S2
25                            subscript_list(A1) ← S1

26          for E2 ← first_aliased_entry(E1) to last_aliased_entry(E1)
27                for T1 ← first_type(E1) to last_type(E1)
28                      for T2 ← first_type(E2) to last_type(E2)
29                            for A1 ← first_access(T1) to last_access(T1)
30                                  S1 ← subscript_list(A1)
31                                  subscript_list(A1) ← NIL
32                                  for A2 ← first_access(T2) to last_access(T2)
33                                        S2 ← subscript_list(A2)
34                                        subscript_list(A2) ← NIL
35                                        if is_read(A1) ∧ is_write(A2)
36                                              TEST_DRIVER(A1, A2, ANTI, FLOW)
37                                        if is_write(A1) ∧ is_read(A2)
38                                              TEST_DRIVER(A1, A2, FLOW, ANTI)
39                                        if is_write(A1) ∧ is_write(A2)
40                                              TEST_DRIVER(A1, A2, OUTPUT, OUTPUT)
41                                        subscript_list(A2) ← S2
42                                  subscript_list(A1) ← S1
```

Figure 5.2: Algorithm for Intersecting References in Access Table.

under the same variable entry data structures are implicitly considered to be aliased, and subscript information cannot be directly compared during data dependence analysis. Lines 11–25 intersect references from the same variable entry and different variable type. Note that lines 13–14 and 16–17 accomplish the removal of all subscript expressions for both aliased accesses to prevent the direct comparison of subscripts, and lines 24–25 reattach the subscript expressions after the intersection process is complete. This process eliminates the constraints for the data dependence problem generated by the subscript expressions and results in conservative dependence analysis for aliased variable references. Although the subscript expressions for variable references of different data types but the same type size could be directly compared, no machine-specific information about simple type sizes is available for the *Pcode* module at this time.

Section 4.2 discussed the conditions for explicit aliasing of two variable entry data structures using the alias graph. Since information about the amount of overlap is unavailable, the subscript information for explicitly aliased variables again cannot be used to calculate the data dependences. Lines 26–42 intersect references from different variable entries, while lines 30–31, 33–34, and 41–42 accomplish the removal and reattachment of subscript expressions for the same reason as described above.

The algorithm also contains several refinements to insure that no two accesses are intersected more than once with the same dependence type. Lines 4 and 11 accomplish this duplication prevention by careful selection of the loop bounds. Although not specifically illustrated, the macros *first_aliased_entry* and *last_aliased_entry* on line 26 provide

a mechanism to avoid traversing the same undirected alias arc in both directions. Recall from Section 4.1.3 that a static reference may actually represent both a read and a write access (for static references on the left-hand side of the compound assignment operators, for example). Therefore, the final condition for the **if** statement on line 5 insures that the subroutine calls on lines 6 and 8 don't produce duplicate intersections for this case.

5.4.2 Omega Test driver

The TEST_DRIVER subroutine prepares and sends a pair of program references to the Omega Test, which generates the data dependence information between that pair. As shown in Figure 5.3, the subroutine takes as arguments a pair of accesses, and the types of the possible dependences from the first to the second access and from the second to the first.

The first two lines of the algorithm prevent dependences between index variable initialization accesses and read accesses of the index variable inside its corresponding loop. These dependences are not calculated since high-level transformations typically do not rely on them to guarantee correctness. However, low-level scheduling and optimizations may require them since the accesses involving the index variable are no longer ordered by the semantics of the *Parloop*.

Loop nesting depths for each access and the common enclosing loop nesting depth for both accesses are calculated in lines 3–5 of the algorithm. *Loop_nesting_depth* is a macro which merely accesses the correct field of the variable access data structure generated

```
TEST_DRIVER(A1, A2, T12, T21)
1     if (T12 = FLOW ∧ is_index(A2)) ∨ (T12 = ANTI ∧ is_index(A1))
2           return
3     D1 ← loop_nesting_depth(A1)
4     D2 ← loop_nesting_depth(A2)
5     D12 ← COMMON_LOOP_NESTING_DEPTH(A1, A2)
6     S1 ← first_subscript(A1)
7     S2 ← first_subscript(A2)
8     while S1 ≠ NIL ∧ S2 ≠ NIL
9           S1 ← next_subscript(S1)
10          S2 ← next_subscript(S2)
11    if S1 ≠ NIL ∨ S2 ≠ NIL ∨ entire_extent(A1) ∨ entire_extent(A2)
12          S1 ← subscript_list(A1)
13          S2 ← subscript_list(A2)
14          subscript_list(A1) ← NIL
15          subscript_list(A2) ← NIL
16    else
17          S1′ ← first_subscript(A1)
18          S2′ ← first_subscript(A2)
19          N ← 1
20          while S1′ ≠ NIL ∧ S2′ ≠ NIL
21                if (D12 = 0 ∧ (func_mod_var(S1′) ∨ func_mod_var(S2′))) ∨
                     (D12 ≠ 0 ∧ (loop_nest_mod_var(S1′) ∨ loop_nest_mod_var(S2′)))
22                      AE1[N] ← affine_expr(S1′)
23                      AE2[N] ← affine_expr(S2′)
24                      affine_expr(S1′) ← NOT_AFFINE
25                      affine_expr(S2′) ← NOT_AFFINE
26                else
27                      AE1[N] ← NIL
28                      AE2[N] ← NIL
29                S1′ ← next_subscript(S1′)
30                S2′ ← next_subscript(S2′)
31                N ← N + 1
32    OMEGA_TEST(A1, A2, T12, T21, D1, D2, D12)
33    if S1 ≠ NIL ∨ S2 ≠ NIL ∨ entire_extent(A1) ∨ entire_extent(A2)
34          subscript_list(A1) ← S1
35          subscript_list(A2) ← S2
36    else
37          S1′ ← first_subscript(A1)
38          S2′ ← first_subscript(A2)
39          N ← 1
40          while S1′ ≠ NIL ∧ S2′ ≠ NIL
41                if AE1[N] ≠ NIL ∨ AE2[N] ≠ NIL
42                      affine_expr(S1′) ← AE1[N]
43                      affine_expr(S2′) ← AE2[N]
44                S1′ ← next_subscript(S1′)
45                S2′ ← next_subscript(S2′)
46                N ← N + 1
```

Figure 5.3: Algorithm for Preparing Reference Pairs for the Omega Test.

during variable reference analysis. The algorithm COMMON_LOOP_NESTING_DEPTH is not given, since it is relatively trivial given that each access contains a pointer to its nearest enclosing *Parloop* and each *Parloop* data structure contains loop nesting depth and parent *Parloop* fields, as described in Section 5.2.2.

Lines 8-15 of the algorithm perform temporary removal of all subscript expressions for both accesses if certain conditions hold, as given on line 11. The first pair of conditions specifies that if the two accesses have a different number of subscripts, then their subscripts cannot be used by the Omega Test to constrain the data dependence problem. (The **while** loop on lines 8–10 iterates through the subscript expressions of both accesses simultaneously until the end of either list is reached.) For example, if one of the accesses is a single-dimensional access of a single array element, and another access is to the entire array (e.g., the array pointer being passed to a function as in call by reference), then subscripts cannot be compared pairwise between the two references, and the only option is to eliminate them from consideration. The second pair of conditions specifies that if the extent of either access is not constrained to a single array element, as explained in Section 4.1.3, then none of the subscripts may be used for the data dependence test. Lines 12-15 accomplish the actual removal and temporary storage of the unused subscripts.

If the subscripts are not temporarily removed, lines 17-31 perform the necessary preparations for the subscripts containing modified variables. The while loop on line 20 iterates through the subscripts for accesses *A1* and *A2* simultaneously using the variables *S1′* and *S2′*, respectively. If the common enclosing loop-nesting depth is zero, then either

subscript containing variables modified in any region in the function (see Section 4.4.2) will render both subscripts useless to the Omega Test, which contains no method to correctly handle modified variables (other than index variables). Although using the entire function as a region is overly conservative, strict determination of whether the modified variable is updated along any path between the two accesses seems unwarranted, since most high-level transformations apply to single-loop nests instead of top-level function statements or adjacent loop nests. However, if the accesses are within a common loop nest, the pair of subscripts being examined must be ignored only if either of them contains a variable modified in that loop nest. Although the choice of the loop nest as a region is again overly conservative, the Omega Test provides no mechanism to handle a more specific region. In either case described above, the corresponding subscripts for each access are essentially ignored by temporarily assigning them a non-affine status, as shown in lines 22-25. The Omega Test then ignores these non-affine subscript pairs when generating data dependence analysis constraints.

Line 32 invokes the Omega Test itself, passing the accesses, dependence types, and loop nesting depths as arguments. The remaining lines of the algorithm restore the removed subscripts or affine expressions back to the pair of accesses.

5.4.3 Data dependence graph representation

As the Omega Test detects each data dependence, it calls an interface subroutine which stores the data dependence information in the data dependence graph. The data

dependence graph consists of the variable access data structures which represent graph nodes, and the data dependence data structures which represent the arcs between the nodes. For convenience, the data dependence data structures will be referred to as dependence arcs, and the variable access data structures as access nodes. Since the data dependence graph contains data structures for both nodes and arcs, the dependence arcs are linked to the access nodes using three sets of pointers. First, each dependence arc contains source and destination pointers to the corresponding access nodes at the tail and head, respectively, of the data dependence arc. Next, each access node contains pointers to a pair of unordered arc lists, one for incoming arcs and the other for outgoing arcs. Finally, the dependence arcs contain pointers to construct the links for this pair of unordered arc lists. This form of graph data structure allows easy traversal of the nodes and connecting arcs.

The data dependence abstraction utilized is a combination of dependence distance and direction vectors. The distance vector gives the difference in index variable values for each enclosing *Parloop* and is represented by an array of signed integers such that the number of elements equals the number of enclosing *Parloops*. Since the Omega Test does not enumerate the possible solutions for non-uniform data dependences, only one distance vector can be generated for each call to the Omega Test. Consequently, if more than one possible distance vector exists, the Omega Test returns a single distance vector, such that the vector entries which can take on multiple values are given a special value signifying an "unknown" distance. However, a single call to the Omega Test may generate

several instances of direction vectors, each representing a different loop which carries the data dependence. In this case, each direction vector is reported as a separate dependence arc. Direction vectors are represented in the dependence arc as bit vectors containing up to six fields of three bits each, one bit for each possible direction. Macro functions are provided to manipulate and test the direction vectors. In addition, the number of enclosing *Parloops* is also recorded in the dependence arc to identify which entries of the distance and direction vectors are utilized.

An additional entry, the data dependence threshold [11], is provided in the dependence arc and represents the number of loop iterations between the source and destination of the dependence for the loop which carries the dependence. Note that the data dependence threshold can be obtained by dividing the distance by the step for the loop which carries the dependence. If the distance for the carrying loop is unknown, the dependence threshold is marked with the same "unknown" value. If the data dependence is not loop carried, then the threshold value is zero.

Finally, a 32-bit vector is provided for marking dependences for transformations. The meaning assigned to the fields in this vector is left to the transformation author's discretion.

6. CONCLUSIONS

This thesis has described, in detail, the variable reference analysis and data dependence analysis packages that generate data dependence relationships for *Fortran* programs. Although solution of the mathematical equations of the data dependence problem is essential for data dependence analysis of *Fortran* programs, correct and efficient program variable and loop analysis must be employed to obtain accurate data dependence relationships. Variable aliasing analysis and modified variable detection are essential components for interfacing *IMPACT* data dependence analysis to data dependence testing software. Furthermore, correct delinearization of array references is important to obtain accurate data dependence relationships for programs with linearized array references. The *IMPACT* data dependence analyzer provides the requisite algorithms for generating accurate data dependence relationships for real *Fortran* programs, enabling compiler writers to implement valid program transformations and optimizations.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, (Toronto, Canada), pp. 266–275, June 1991.

[2] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, August 1992.

[3] U. Banerjee, *Dependence Analysis for Supercomputing.* Boston, MA: Kluwer Academic Publishers, 1988.

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

[5] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York, NY: ACM Press, 1991.

[6] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations.* Boston, MA: Kluwer Academic Publishers, 1993.

[7] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," in *Proceedings of the ACM SIGPLAN '90 Symposium on Compiler Construction*, pp. 53–65, 1990.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[9] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the Omega Test," in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151, June 1992.

[10] W. Pugh and D. Wonnacott, "Going beyond integer programming with the Omega Test to eliminate false data dependences," Tech. Rep. CS-TR-2993, Department of Computer Science, University of Maryland, College Park, December 1992.

[11] W. Pugh, "The definition of dependence distance," Tech. Rep. CS-TR-2992, Department of Computer Science, University of Maryland, College Park, November 1992.

[12] M. Wolfe, "Experiences with data dependence abstractions," in *Proceedings of the 1991 International Conference on Supercomputing*, pp. 321–329, June 1991.

[13] P. P. Chang, "The Hcode language and its environment." IMPACT compiler documentation, Center for Reliable and High-performance Computing, University of Illinois, Urbana, IL, 1989.

[14] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Englewood Cliffs, NJ: Prentice Hall, 1978.

[15] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1988.

[16] N. J. Warter, P. P. Chang, S. Anik, G. E. Haab, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Charlie C: A reference manual." IMPACT compiler documentation, Center for Reliable and High-performance Computing, University of Illinois, Urbana, IL, 1991.

[17] American National Standards Institute, New York, NY, *American National Standard Programming Language FORTRAN*, 1978. ANSI X3.9-1978.

[18] S. I. Feldman, D. M. Gray, M. W. Maimore, and N. L. Schryer, "A Fortran-to-C converter," Computing Science Tech. Report 149, AT&T Bell Laboratories, Murray Hill, NJ, June 1990.

[19] N. J. Warter and G. E. Haab, "Pcode manual." IMPACT compiler documentation, Center for Reliable and High-performance Computing, University of Illinois, Urbana, IL, 1991.

[20] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[21] K. Subramanian, "Loop transformations for parallel compilers," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[22] S. Anik, "Architectural and software support for executing numerical applications on high performance computers," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[23] Y. Yamada, "Data relocation and prefetching for programs with large data sets," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[24] P. Chang, "Compiler support for multiple instruction issue architectures," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[25] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for C programs," *Software Practice and Experience*, vol. 22, pp. 349–370, May 1992.

[26] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[27] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1992.

[28] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[29] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," Tech. Rep. CRHC-91-18, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[30] R. A. Bringmann, "Enhancing instruction level parallelism through compiler-controlled speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[31] N. J. Warter, "Modulo scheduling with isomorphic control transformations," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[32] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[33] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[34] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[35] W. Y. Chen, "An optimizing compiler code generator: A platform for RISC performance analysis," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 1991.

[36] R. G. Ouellette, "Compiler support for SPARC architecture processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[37] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA 94303, February 1994.

[38] V. Maslov, "Delinearization: An efficient way to break multiloop dependence equations," in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 152–161, June 1992.