ENABLING DEPENDENCE ANALYSIS IN C

by

Ian Christopher Maione

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

© Copyright by Ian Christopher Maione 1997
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-28766-1
Abstract

Enabling Dependence Analysis in C

Ian Christopher Maione
Master of Science
Graduate Department of Computer Science
University of Toronto
1997

Dependence analysis is a fundamental tool used in many compiler transformations which optimize and parallelize scientific code written for high-performance vector and parallel computer architectures. Implementing dependence analysis for the C programming language is difficult because of complications caused by nonstandard control flow and use of pointers to reference arrays.

In order to enable dependence analysis in C, code can be preprocessed to convert loops which violate FORTRAN-like conventions into a canonical form which can be processed successfully by the dependence analyzer. We developed two algorithms to enable such processing. Loop control flow normalization (LCFN) normalizes loop control flow and array reference subscripts. Pointer array access normalization (PAN) recovers implicit array references through pointers.

A prototype implementation of the LCFN and PAN methods was built using the Parafrase-2 compiler. Experimental results generated using the SPEC95 and NAS benchmark suites showed that these techniques can successfully enable dependence analysis.
Acknowledgements

First of all, I would like to thank my supervisor, Professor Tarek Abdelrahman, for his helpful support and advice in the development of this thesis. I would also like to thank my second reader, Professor Ken Sevcik, for his helpful advice on improving the quality and presentation of this thesis.

I would also like to thank my fellow students in the zoo, for the enjoyable work environment they have provided during the writing of this thesis. I particularly would like to thank the co-creators of DAWG, Jin Lee and Anuj Gujar, for helping to make possible the endless hours of recreation without which this thesis would never have been completed. In this vein, I would also like to thank Daniel Marcu, for keeping me humble by defeating me day after day. I would like to thank Francois Pitt for his help and advice in dealing with the intricacies of \LaTeX{}. Rich Paige for ensuring that I never went into sports-withdrawal. Angela Demke for briefly putting up with me as an officemate, and Jeff Tupper for putting up with me as an officemate for much longer.

I particularly would like to express my appreciation for the loving support and encouragement of my family, especially my parents, which has been invaluable to me during the time that I have been completing this work.

Finally, I would like to gratefully acknowledge the financial support provided by NSERC and the University of Toronto for the development of this thesis.
Contents

1 Introduction
   1.1 The Dependence Analysis Problem ........................................ 2
      1.1.1 The Dependence Problem in FORTRAN ................................. 4
      1.1.2 The Dependence Problem in C ...................................... 9
   1.2 Admissible Loops .................................................................. 7
      1.2.1 Admissible Loop Normalization ..................................... 8
   1.3 Thesis Contributions ................................................................ 9
   1.4 Thesis Organization ................................................................ 10

2 Normalizing Loop Control Flow .................................................... 11
   2.1 LCFN Algorithm Overview ..................................................... 13
   2.2 Loop Syntax Preprocessing ................................................... 16
   2.3 Computing Loop Trip Counts ................................................ 18
      2.3.1 Handling Zero Trip Loops .............................................. 21
   2.4 Subscript Normalization ...................................................... 24
   2.5 Canonical Loop Generation .................................................. 25

3 Pointer Array Access Normalization ............................................. 27
   3.1 PAN Algorithm .................................................................. 29
      3.1.1 One-Dimensional PAN ............................................... 31
      3.1.2 Multidimensional PAN ............................................... 32

4 Prototype Implementation ............................................................ 36
   4.1 The Parafrase-2 Compiler .................................................... 36
4.1.1 Overview of Parafrase ........................................ 36
4.1.2 LCFN Implementation ........................................ 40
4.1.3 PAN Implementation ........................................ 40
4.2 Experimental Evaluation ....................................... 41
  4.2.1 Experimental Procedure ................................. 41
  4.2.2 Experimental Results .................................. 43

5 Related Work ....................................................... 48

6 Conclusion and Future Work ................................... 53
  6.1 Conclusion .................................................. 53
  6.2 Future Work ................................................ 54

A Induction Variable Analysis .................................... 56

Bibliography ............................................................ 61
Chapter 1

Introduction

*Dependence analysis* is a fundamental tool used in many compiler transformations which have been developed to optimize scientific and numerical code written for high-performance vector and parallel computer architectures. Transformations which attempt to maximize parallelism and/or memory locality typically require dependence analysis, and their efficacy is often directly related to the efficacy of the latter analysis. Bacon et al. summarize a number of such techniques [BGS94]. Because dependence analysis is such a fundamental part of compilation for parallel machines, a number of different techniques have been developed for analyzing dependences between array references in loop nests. [Tow76, Wol89, Ban88, BCK79, GKT91, LYZ90, WT92, Pug92]. Dependence analysis techniques have been implemented in several research compiler systems [AK84, ZBG88], and have also been implemented in some commercial compilers such as KAP and VAST.

Implementations of dependence analysis in both research and commercial systems have focused on FORTRAN programs. Since most scientific code has been written in FORTRAN, this is a natural development. By contrast, very few compilers are capable of doing dependence analysis in the C programming language. There is no advantage to writing scientific code in C if compilers cannot effectively parallelize it, because of the more varied syntax of the C language. Conversely, since scientific applications coded in C are rare, the need for C compilers to do sophisticated dependence analysis has not existed. However, the growing use of C++ as a language suggests that the ability to handle C constructs may be a requirement for future compiler systems which do
In order to implement dependence analyzers which are capable of handling C effectively, one can either handle the complexities of C syntax within the dependence analyzer itself, or one can attempt to preprocess the C source code before dependence analysis in order to make it better adhere to the form of FORTRAN-style loops. The latter approach is clearly better from the point of view of modularity, since the particular syntactic structures of a programming language are not essential to any particular dependence analysis algorithm. It is also attractive from a software engineering point of view, since handling C constructs outside the dependence analyzer allows implementors to avoid redesigning and rewriting analyzer code each time a new dependence analysis technique is implemented. As a first step toward this goal, this thesis applies a number of different compiler techniques to the problem of generating C code which is more amenable to standard dependence analysis methods, and implements them within an existing compiler environment (Paraphrase-2). We term the process of generating this normalized code *admissible loop normalization*.

The remainder of this chapter is organized as follows: Section 1.1 outlines the general dependence analysis problem. In Section 1.1.1 and Section 1.1.2, the dependence analysis problem is outlined in terms of the FORTRAN and C languages respectively, and the difficulties introduced by the C language are described. In Section 1.2 a normalized form for C loops is defined, as a goal for admissible loop normalization.

## 1.1 The Dependence Analysis Problem

Dependence analysis is a well-studied problem, and is fundamental to many other compiler analyses designed to parallelize scientific and numerical programs automatically. A *data dependence* is said to exist between two statements in a loop nest if one statement writes a value that the other statement uses. There are three different types of data dependence which are relevant in the context of parallelization. A *flow dependence* exists between statements $S_1$ and $S_2$ if $S_1$ writes a value which is later read by $S_2$. Similarly, an *anti-dependence* exists if $S_1$ reads a value which is later used by $S_2$. An *output de-
Figure 1.1: The dependence analysis problem

\[
\begin{align*}
\text{for } l_1 &= l_1 \text{ to } u_1 \text{ do} \\
&\quad \text{for } l_2 = l_2 \text{ to } u_2 \text{ do} \\
&\quad \quad \vdots \\
&\quad \quad \text{for } l_d = l_d \text{ to } u_d \text{ do} \\
&\quad \quad \quad \Delta(f_1(l_1, \ldots, l_d), \ldots, f_d(l_1, \ldots, l_d)) \\
&\quad \quad \quad \Delta(g_1(l_1, \ldots, l_d), \ldots, g_d(l_1, \ldots, l_d)) \\
&\quad \quad \text{end for} \\
&\quad \vdots \\
&\text{end for} \\
&\text{end for}
\end{align*}
\]

A dependence exists if both \( S_1 \) and \( S_2 \) write the same value. Parallelizing compilers generally attempt to execute different iterations of loops in parallel. Since a dependence represents a serial semantic relationship between statements, a parallelizer must be able to detect dependences, in particular loop-carried dependences, which exist between different loop iterations. In the absence of dependence analysis, a compiler cannot parallelize, since executing parallel iterations in the presence of dependences can lead to incorrect code.

In general terms, the dependence analysis problem can be formulated as described by Wolfe and Tseng [WT92]. Given a loop nest as in Figure 1.1, a dependence test attempts to determine if different array references can access the same array element during the execution of the loop nest, i.e., that there exist two sets of values \( \{l_1 = i_1, \ldots, l_d = i_d\} \) and \( \{l_1 = j_1, \ldots, l_d = j_d\} \) such that:

1. \( l_k \leq i_k \leq u_k \) for all \( 1 \leq k \leq d \)
2. \( l_k \leq j_k \leq u_k \) for all \( 1 \leq k \leq d \)
3. \( f_m(i_1, \ldots, i_d) = g_m(j_1, \ldots, j_d) \) for all \( 1 \leq m \leq s \), where \( s \) is the number of array subscript functions.

The above set of equations are known as the dependence equations. This formulation of the dependence analysis problem implicitly assumes that the semantics of the pseudocode for constructs have certain properties.
Figure 1.2: Loop nest in FORTRAN

\begin{verbatim}
DO 100 I1 = l1,ul
   DO 200 I2 = 12,u2
      : 
      DO 300 Id = ld,ud
         A(a1 * I1 + ... + ad * Id) = ...
         ... = A(b1 * I1 + ... + bd * Id)
   300      CONTINUE
   200     CONTINUE
100 CONTINUE
\end{verbatim}

1. Each loop has a standard syntactic form, with an explicitly defined index variable \( I \) which is not modified by statements within the loop nest (other than loop control statements).

2. Each of the loops has explicit iteration limits \( l_k \) and \( u_k \), which are not modified within the loop nest (some dependence analysis techniques may require these to be literal constants, or they may be symbolic expressions, but they must at least be loop invariant).

3. Each array reference is made using an expression specifying the name of a statically declared \( k \)-dimensional array, as well as \( k \) array index expressions. Each array index is a function solely of the loop indices (i.e., the functions \( f_1, \cdots, f_s, g_1, \cdots, g_s \) are functions only of \( I_1, \cdots, I_d \)), as well as possibly constants or loop invariants.

4. There is no irregular control flow in the loop (i.e., there are no statements within the loop body which branch outside the loop).

1.1.1 The Dependence Problem in FORTRAN

In FORTRAN, the dependence problem can be expressed using a DO loop nest, as illustrated in Figure 1.2. The FORTRAN syntax corresponds well to the pseudocode formulation of the dependence problem, since the semantics of a DO loop stipulate that the values of the initial and upper limits of each loop are the values of the corresponding
expressions at the beginning of the loop's execution, regardless of whether they are later modified within the loop. The index variables are not modified within the loop, and each access to an array is made through an explicit array reference consisting of the name of a static array and an explicit array index expression. Furthermore, there is a well-defined loop increment (in this case 1), which also does not change within the loop. Most importantly, the index variables themselves, their lower and upper limits, and the loop increment can be determined by simple examination of the program syntax, without recourse to further analysis. Thus a dependence analyzer for FORTRAN code does not usually need complex analysis of the loop to construct the dependence equations.

It should be noted that it is possible for aliasing to become an issue in FORTRAN through the use of COMMON blocks. A COMMON block can cause a global variable or array to be referenced by different names inside subroutines. Even this type of simple aliasing can cause serious problems for an optimizing compiler; nevertheless, the situation is not nearly as complicated as in C, where pointer relationships can be dynamically changed at runtime through the use of pointer variables.

1.1.2 The Dependence Problem in C

Implementing dependence analysis in C is substantially more complex, because of the wider range of loop structures which exist in C, as well as the freer semantics which C allows the programmer in regards to loop control, and access to arrays. These aspects of the C language can be divided into two broad categories:

- Loop control issues.
- Pointer access issues.

Loop Control Issues

There are several aspects to C loop control flow which can violate the assumptions described in Section 1.1:

\footnote{Having these values means that the total count of the loop can be determined, which is important in several dependence analysis methods.}
1. The loop can be written with various types of statements. (for, while, do-while, if/goto).  

2. C allows for and while loops to have free syntax with respect to loop index variables, i.e:

- C does not require that a for loop have an explicit index variable. There is no placeholder in a while loop for an index variable.
- A for or while loop may have multiple index variables, either explicitly or implicitly.
- C allows index variables defined in a for statement to be arbitrarily modified within the body of the loop.
- C does not require that a for statement have an explicit loop increment statement. Similarly, there is no placeholder in a while statement for a loop increment.

Because of these considerations, dependence analysis in C is much more difficult than in FORTRAN, because the basic information which the analyzer needs in order to apply many of the common dependence analysis techniques is no longer immediately available from the source code.

**Pointer Related Issues**

Another significant complication for a dependence analyzer in C arises from the fact that variables and arrays can be and often are accessed using pointers. The mere presence of pointers in a program can make many kinds of analysis substantially more complex. This is due to the fact that many compiler analyses require conservative assumptions if certain variable references cannot be analyzed. In the case of dependence analysis, pointers pose difficulties which can be separated into two general categories:

1. Modification of scalars by pointer dereference.

---

2Some of these structures are also possible in FORTRAN, but are more commonly used in C.
Since the dependence analyzer is trying to determine whether sets of array index expressions are equivalent over the loop iteration space, a dereference through a scalar pointer variable could potentially affect any of the loop control variables or variables which are involved in array index expressions. In such a case, the dependence analyzer may have to assume dependence in the absence of more specific information, limiting the efficacy of the analysis. Although the general problem of alias analysis in C is an extremely complex one, even partial resolution of scalar pointer references would make C more amenable to dependence analysis.

2. Access to arrays via pointer dereference.

In C, even determining what the index expressions of an array reference are can be a difficult problem because of the constructs in C which allow the programmer to access array structures. For example, C allows the programmer to use pointer arithmetic to access arrays without using explicit array index expressions. This type of syntax can obscure the array elements being accessed from the compiler. The situation is further complicated by the fact that pointer aliasing can even obscure which array is being accessed in a given reference. Once again, in the absence of appropriate information, the conservative assumption of dependence must be made in these cases.

1.2 Admissible Loops

In order to enable dependence analysis in C, it is reasonable to attempt to transform C programs which violate the previously described simplifying assumptions into programs which conform to them. Thus, we can define our goal as follows:

**Definition 1** A *canonical* loop is a loop of the form:

```c
for(i = 0; i < \Phi; i = i + 1)
{
    (loop body)
}
```
such that:

(i) \(i\) is an `int` variable which is the index variable for the loop. \(i\) is not used outside the loop, and is initialized to 0 in the `for` statement. Furthermore, \(i\) is not modified by any statement inside the loop body.

(ii) There is no irregular control flow within the loop (i.e. `break`, `continue`).

(iii) There is an explicit loop exit test of the form \((i < \Phi)\). \(\Phi\) may be any C expression, but it may not have side effects, and its value must also be loop invariant. Since the index variable \(i\) is initialized to 0, the expression \(\Phi\) directly represents the trip count of the loop.

(iv) There is an loop increment statement inside the `for` statement. The value of the increment is 1 and does not change during execution of the loop.

(v) Each access to an array \(a\) in the loop is of the form:

\[ a[E_1][E_2]\ldots[E_k] \]

where \(E_1, E_2\ldots E_k\) are all expressions which involve only \(i\). loop invariant expressions, or constants.

A canonical loop describes our goal; a loop which closely resembles the semantics of a FORTRAN-style DO loop, and from which required information for dependence analysis can be extracted from the source code itself. Thus, following the terminology introduced by Justiani and Hendren [JH94], we define:

**Definition 2** Given an arbitrary C loop \(L\). \(L\) is admissible if there is a canonical loop \(L'\) which is semantically equivalent to \(L^3\).

### 1.2.1 Admissible Loop Normalization

In order to enable dependence analysis for C, we wish to be able to transform as many loops as possible into equivalent canonical forms. We term this transformation admissible

\[^3\text{Note that strictly speaking, this is a statement about the semantics of a certain piece of C code, regardless of whether a compiler can determine that such a canonical loop exists.}\]
loop normalization. This transformation attempts to (a) determine that a given loop is indeed admissible, and (b) construct the appropriate canonical form for that loop. To this end, a compiler pass has been implemented within the Parafrase-2 compiler environment to generate canonical forms in C for some types of admissible loops.

The Parafrase-2 parallelizing compiler [PGH90] is a vectorizing/parallelizing compiler which operates as a source-to-source translator. Parafrase-2 can compile FORTRAN or C code, and represents source code in an intermediate form which can then be manipulated by various passes. This intermediate form contains sufficient information to reconstruct C source code after analyses or transformations have been applied. The core compiler contains passes implementing a number of existing analyses and transformations, including flow graph construction, code generation, constant propagation, induction variable substitution, dead code elimination, etc. Parafrase-2 itself is implemented in the C programming language, and contains methods which can be used to access the internal representation and implement new passes. Admissible loop normalization has been implemented on top of the existing induction elimination pass in Parafrase-2.

1.3 Thesis Contributions

The primary contributions of this thesis are the development of algorithms for normalizing C code which does not conform to the requirements of dependence analysis, with respect to the loop statement (the presence of an explicit loop index variable and for statement, and an explicit loop trip count expression), the format of array index expressions, and the use of pointers to access arrays. For this purpose, existing compiler analysis techniques have been applied to the problem, and existing techniques for computing loop trip counts have been extended. This thesis also shows how these techniques may be implemented within a real compiler environment, and experiments with existing parallel benchmark programs are used to demonstrate that these techniques are able to successfully enable dependence analysis for real C programs.
1.4 Thesis Organization

This thesis is organized in six chapters. Chapter 1 introduces the dependence analysis problem, the difficulties which arise in the C language with respect to dependence analysis, and defines a goal for enabling dependence analysis in C. Chapter 2 describes loop control flow normalization, which is an algorithm for normalizing C loops in the absence of pointer operations. Chapter 3 describes pointer array access normalization, which is an algorithm for normalizing certain types of C pointer operations. In Chapter 4 an implementation of these algorithms is described within the Parafrase-2 compiler environment and experimental results are presented showing the efficacy of these methods for selected benchmark programs. Chapter 5 describes related work, reviewing other work done on the C dependence analysis problem, as well as important background material on induction variable analysis and alias analysis. Finally, Chapter 6 presents conclusions and possible future extensions.
Chapter 2

Normalizing Loop Control Flow

In this chapter we will consider the problem of admissible loop normalization under some simplifying assumptions. In particular, we will consider C programs which do not have pointer references in them, either to scalars or to arrays. These restrictions will be eased in Chapter 3. We will also restrict the scope of the analysis to the intraprocedural level, so that there are no function calls that affect non-local variables, and no recursive function calls. Considering only these types of programs allows us to focus on issues relating to loop control flow.

Figure 2.1 illustrates a source program that is unnormalized for dependence analysis. The loop in Figure 2.1(a) exhibits several characteristics which do not conform to the definition of a canonical loop (see Section 1.2). We want to be able to take such a loop and generate an equivalent canonical loop (such as the one in Figure 2.1(b)). In order to do this, there are several aspects of the input loop which must be handled:

(a) Loop syntax.

Loops can be written in different syntactic forms. For example, the loop in Figure 2.1(a) is written with if and goto, instead of with for. Similarly, one can write loops using other control structures (while, for, do-while, etc.). Generating a canonical form entails expressing a loop as a for loop regardless of its syntactic form, as in Figure 2.1(b). It is also necessary to associate an explicit index variable with the loop. The variable ix in the normalized loop serves this purpose, whereas the if/goto form has no such explicit variable. In C, only the for statement
Figure 2.1: Nonstandard Loop Control Flow

| i = 0; | for(ix=0; ix < ceil((100+k)/(2+r);ix++) |
| N = 100; | { |
| j = 0; | a[(r + 4) * ix] = 5; |
| L1: if((i+j) >= N+k) | } |
| goto L2; | |
| v = j + (2*i); | |
| a[v] = 5; | |
| i = i + 1; | |
| j = j + r; | |
| i = i + 1; | |
| goto L1; | |
| L2: | |

(a) input loop (b) normalized loop

has a specific placeholder for a loop index variable, and even in this case it is not syntactically required.

(b) Loop trip count computation.

An important aspect of determining admissibility is the computation of a trip count for the loop. If the compiler can generate a loop invariant expression representing the number of iterations of the loop, the for statement of the canonical form can be straightforwardly generated by the compiler. To do this, the compiler must use induction variable analysis in order to attempt to derive a quantity for the trip count based on the type of exit condition for the loop. In Figure 2.1, the compiler must be able to determine that the variables i and j are induction variables for the loop, and subsequently compute an expression for the loop trip count: \( \text{ceil}((100 + k) / (2 + r)) \).

(c) Subscript normalization.

Given that each array reference has explicit array index expressions for each di-
mension, the compiler must re-express each of them in terms of the loop index variable, since these expressions may not necessarily be written by the programmer in terms of the loop index variable. For example, a programmer may use an induction variable in a loop to avoid having a linear array index recomputed on each iteration of a loop. While this is desirable when compiling for a serial machine, it can hinder dependence analysis, and therefore parallelization. Furthermore, there are many different equivalent ways of writing polynomial functions syntactically due to the properties of commutativity, distributivity, and associativity. The array index in Figure 2.1(a) is expressed in terms of an induction variable $v$, and is re-expressed in terms of the index variable $ix$ in the normalized loop. In general, we would like to be able to express each array index expression as a standard form $a_0 + a_1 * ix_1 + \cdots + a_n * ix_n$, where $a_0, \ldots, a_n$ are loop invariant expressions and $ix_1, \ldots, ix_n$ are enclosing loop index variables.

The remainder of this chapter is organized as follows. In Section 2.1, an overview of the algorithm used to normalize loop control flow is presented. Each of the primary phases in the algorithm is then described in the following sections. Section 2.2 describes loop preprocessing. Section 2.3 describes the computation of trip counts for loops. Section 2.4 describes subscript normalization. and finally Section 2.5 describes the generation of canonical loop forms.

## 2.1 LCFN Algorithm Overview

The algorithm for loop control flow normalization (LCFN) can be described at a high level in terms of the three aforementioned phases: loop syntax, trip count computation, and subscript normalization. These three phases prepare a loop for canonical loop generation, in which the loop is replaced with a for loop satisfying the conditions described in Section 1.2. Figure 2.2 summarizes the steps involved. We can consider the operation of this algorithm on the example given in Figure 2.3.

Figure 2.3(a) shows the original loop from Figure 2.1 expressed as a for loop\(^1\) The

\(^1\)The LCFN algorithm assumes input loops are either in while or for form.
**Figure 2.2: LCFN algorithm overview**

**Input:** A C function \( f \) which satisfies the following:

(a) \( f \) does not contain function calls that have side effects on variables appearing within \( f \).

(b) There are no assignments or references through pointer variables in \( f \).

(c) All loops in \( f \) are either \texttt{while} or \texttt{for} statements.

(d) There are no \texttt{goto} statements in \( f \).

1. for each loop \( L \) in the program
2. preprocess \( L \) for analysis
3. if \( L \) violates conditions for analysis
4. mark \( L \) inadmissible
5. continue \quad \{skip loop \( L \}\)
6. analyze \( L \) for induction variables (IV)s
7. compute trip count for \( L \) based on IV analysis
8. if trip count could not be computed
9. mark \( L \) inadmissible
10. continue
11. for each array reference expression \( E \) in \( L \)
12. if \( E \) is an induction expression
13. replace \( E \) by its equivalent induction expression
14. end for
15. generate canonical form for \( L \)
16. replace \( L \) in parse tree by canonical form
17. end for

loop syntax preprocessing phase takes the following steps, whose results are seen in Figure 2.3(b):

(i) The original \texttt{for} loop is converted to an equivalent \texttt{while}.

(ii) A compiler-generated index variable \texttt{ix} is added to the loop. \texttt{ix} is initialized to zero immediately before the \texttt{while} and is incremented by 1 in the last statement of the \texttt{while} loop body.

(iii) The quantity \( T = ((N+k)-(i+j)) \) is added to the loop representing the expression whose value determines when the loop will exit (see Section 2.3).
Induction variable (IV) analysis is then employed to express the quantity $T$, as well as the variable $v$ appearing in the array reference $a[v]$, in terms of the index variable $i_{x1}$. The loop trip count can be computed from this to be $\text{ceil}(\frac{100+k}{2+r})$.

Induction variable analysis is an important and common technique for analyzing the values of variables within loops at compile time, and is crucial to the success of admissible loop normalization. IV analysis involves examining the assignments to variables within a loop in order to discover whether the values assigned to a given variable on each iteration form a sequence which can be described by a closed-form expression in terms of the loop iteration. Those unfamiliar with induction variable analysis are referred to in Appendix A, where both the analysis and the various techniques for doing it are reviewed in detail.

Finally, in Figure 2.3(c), the canonical form is generated for the loop, by moving the
compiler-generated index variable initialization and update into the for statement, and placing the computed trip count expression into the exit test of the for. The computed IV expression for the array access is substituted, and dead code elimination removes the extraneous variables.

2.2 Loop Syntax Preprocessing

In the absence of goto statements, handling loop control flow is considerably simplified. Since programmers tend not to write code using gotos in most cases, and elimination of goto statements is a well-studied problem, this is not a crucial issue. However in many situations the description and implementation of compiler algorithms are greatly simplified by removing them from consideration. In particular, Erosa and Hendren [EH94] describe a goto elimination transformation for C programs. This transformation removes goto statements by replacing them with equivalent structured programming constructs (i.e., while, do-while, etc). Thus, we assume that such a transformation has already been applied before processing begins, and that the loops which are presented to the LCFN analysis are either while or for statements.

Since the subsequent phases of the algorithm assume that loops are in while form, the preprocessing phase first converts any input for loops to while form. If the loop ends up to be an admissible one, then the while will be converted back to the canonical for form at the end of the analysis. Note that for loops can be converted to while form directly [KR88].

In order to further analyze a given loop, the loop must satisfy the following conditions:

- The loop has only one exit, that being the condition appearing in the while statement itself.

- The exit condition appearing in the while statement must be an integer comparison2.

That is, the while statement must have the form

\[ \text{Although the operators } (<, >, \leq, \geq) \text{ are valid for non-integer types, the later induction variable and trip count analysis phases will only be effective for integer variables, so this restriction is made here.} \]
while(\varepsilon), where \varepsilon is an expression of the form:

\[(\varepsilon_1 \ OP \ \varepsilon_2), \varepsilon_1 \text{ and } \varepsilon_2 \text{ are arbitrary } C \text{ expressions, and } OP \text{ is one of the arithmetic comparison operators } (<, >, \leq, \geq).\]

- The exit condition does not contain side effects.

A loop which violates any of these conditions is deemed inadmissible. Given that the above conditions are satisfied, the compiler proceeds by adding an explicit index variable to the loop. At this point, a variable is also added to the loop to hold the *trip count test expression (TCTE)* for the loop (see Section 2.3). The complete preprocessing phase is summarized in Figure 2.4.

Figure 2.4: Loop Syntax Preprocessing

**Input:** A loop L which is either a *while* or *for* loop.

1. if L is a *for* loop
2. let I be the initialization expression of the *for*
3. move I immediately preceding the *for* statement
4. let U be the update expression of the *for*
5. let S be the last statement of the *for* loop body
6. move U to immediately follow S
7. let E be the exit expression of the *for*
8. replace the *for* statement by *while*(E)
9. for each basic block BB in L
10. if BB not= head(L) and BB contains a branch outside L
11. mark L inadmissible
12. if E is not an arithmetic comparison
13. mark L inadmissible
14. if E contains side effects
15. mark L inadmissible
16. if L already marked inadmissible
17. return
18. add index variable ix to symbol table
19. insert \(ix = 0\) before *while* statement
20. insert \(ix = ix + 1\) as last statement of loop body
21. add assignment \(T = TCTE\) as first statement of loop body

Figure 2.5 illustrates an input loop and the state of the loop after preprocessing has been applied.
2.3 Computing Loop Trip Counts

Given that we are dealing with a while loop, which has only a single exit, we wish to be able to compute a trip count for the loop. Under these conditions, the trip count of the loop is the number of loop iterations (possibly zero or $\infty$) which will be executed until the condition inside the while statement becomes false.

Computing a trip count for a while loop is not as straightforward as for a FORTRAN DO loop, because the exit condition can have various forms, and because the variables appearing in the exit condition may be modified inside the loop. Also, the syntax of the while statement does not specify how the increment to the index variable occurs. Figure 2.6 illustrates three equivalent while loops which have different exit conditions.

In order to determine a trip count for a given exit condition, the compiler must be able to analyze the values that the exit condition will take on each iteration of the loop. Wolfe [Wol92] describes how to do this if the loop exit condition is a integer comparison, which can be classified by the compiler as an induction expression. Given a pseudocode exit condition of the form \((\text{if } \varepsilon_1 \leq \varepsilon_2 \text{ exit loop})\) for expressions $\varepsilon_1$ and $\varepsilon_2$, Wolfe's method treats the comparison as a subtraction. Thus the exit condition is treated as equivalent to the condition \((\text{if } \varepsilon_1 - \varepsilon_2 \leq 0 \text{ exit loop})\). Wolfe then computes a trip count if the
Figure 2.6: while loops with nonconstant exit conditions

\[
\begin{array}{|c|c|c|}
\hline
i = 0; & i = 0; & i = 0; \\
N = 100; & N = 200; & v = N - i; \\
while(i <= N) & while(i <= N) & while(v > 0) \\
\{ & \{ & \{ \\
a[i] = 5; & N = N - 1; & a[i] = 5; \\
i = i + 1; & a[i] = 5; & i = i + 1; \\
\} & \} & \} \\
(a) & (b) & (c) \\
\hline
\end{array}
\]

Subtraction \( \varepsilon_1 - \varepsilon_2 \) can be classified by the compiler as a linear induction expression of the form \( \varepsilon_1 - \varepsilon_2 = (\alpha \times ix + J) \). where

(a) \( \alpha \) and \( J \) are integer constants.

(b) \( ix \) is the loop index variable.

The trip count can then be expressed as follows:

\[
\text{trip count} = \begin{cases} 
\infty & \text{if } \alpha > 0 \text{ and } J > 0 \\
0 & \text{if } J \leq 0 \\
\left\lfloor \frac{-J}{\alpha} \right\rfloor & \text{otherwise}
\end{cases}
\]

Since the semantics of a while loop specify that the loop is to be exited when the condition appearing within the while statement itself becomes false, the other integer comparison operators \((<, >, \geq)\) can be handled using the table in Figure 2.7:

The expression in the third column of Figure 2.7 is termed the trip count test expression (TCTE) for the loop. During preprocessing the compiler adds a temporary variable, which is assigned the TCTE at the beginning of each loop iteration. This allows the compiler to analyze it as an induction variable, as if it were any other ordinary program variable.
In practice, there are several other issues that the compiler must deal with. Since several induction variable analysis techniques exist which are capable of detecting and representing nonlinear induction expressions, the compiler must be able to determine whether a given induction expression is linear in the loop index variable and derive the expressions \( \alpha \) and \( \beta \) from it. Furthermore, the compiler should deal with situations in which \( \alpha \) and \( \beta \) are symbolic (but loop invariant) expressions at compile time, since in realistic programs \( \alpha \) and \( \beta \) may not be known constants. The compiler must generate an appropriate trip count expression in terms of \( \alpha \) and \( \beta \) and ensure that it deals with cases involving zero or infinite trip counts reasonably.

Figure 2.7: Trip count test expressions

<table>
<thead>
<tr>
<th>while statement</th>
<th>Positive Exit Condition</th>
<th>Trip Count Test Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(( C ))</td>
<td>if(( C )) exit</td>
<td>if(( C \leq 0 )) exit</td>
</tr>
<tr>
<td>( (E_1 &lt; E_2) )</td>
<td>( E_2 \leq E_1 )</td>
<td>( E_2 - E_1 )</td>
</tr>
<tr>
<td>( (E_1 &gt; E_2) )</td>
<td>( E_2 &gt; E_1 )</td>
<td></td>
</tr>
<tr>
<td>( (E_1 \geq E_2) )</td>
<td>( E_1 \leq (E_2 - 1) )</td>
<td>( E_1 - (E_2 - 1) )</td>
</tr>
<tr>
<td>( (E_1 \leq E_2) )</td>
<td>( E_2 \leq (E_1 - 1) )</td>
<td>( E_2 - (E_1 - 1) )</td>
</tr>
</tbody>
</table>

Note that in Wolfe's method, the expressions \( \alpha \) and \( \beta \) are literal constants, so that the resulting trip count expression is also a constant. When symbolic expressions are involved, the computation of the trip count expression (that is, the expression \( \Phi \) appearing in the canonical for statement) can be described according to one of several cases:

(a) \( \alpha \) and \( \beta \) are constant values known at compile time, in which case we can generate a trip count expression based on Wolfe's formula as described above:

(b) \( \alpha \) and \( \beta \) are symbolically divisible. If \( \alpha \) and/or \( \beta \) are not compile time constants, the compiler may still be able to compute a trip count directly if the symbolic value of \( \alpha \) divides the symbolic value of \( \beta \)\(^3\). Such a situation can occur in a loop such as the following:

\(^3\)Parafrase-2 contains functionality to compute such symbolic divisions where \( \alpha \) and \( \beta \) are polynomial expressions.
\( N = \text{foo}(); \)
\( i = 0; \)
\( ix = 0; \)
\( \text{while}(i < (N * N) + N) \)
\{ 
    \( T1 = ((N * N) + N) - i; \)
    \( i = i + N; \)
    \( ix = ix + 1; \)
\}

In this loop, \( \alpha = -(N) \), and \( J = (N*N+N) \), and so the trip count can be expressed as \( N + 1 \). In general, the result of the symbolic division is used as the trip count expression:

(c) \( \alpha \) and \( J \) are neither constant nor symbolically divisible. In this case, the syntactic trip count will involve an explicit call to the C library \texttt{ceil} function. since \( \alpha \) and \( J \) are not divisible either as constants or symbolic expressions. To generate the trip count expression, the compiler generates the function call \texttt{ceil}(-\( J/\alpha \)).

### 2.3.1 Handling Zero Trip Loops

In cases where the values of \( \alpha \) and \( J \) are not known at compile time, situations in which a loop trip count is zero or \( \infty \) are more difficult to detect. The former are more of a concern, since infinite loops should not be encountered in correct scientific code. and can be considered to be either programmer error or invalid input.

These types of problems can be illustrated by the following programs:

In Figure 2.8(a), the \texttt{while} statement will have a trip count of 0 if \( N \leq 0 \), but a trip count of \( N \) otherwise. In Figure 2.8(b), the \texttt{while} statement will have a trip count of 0 if \( i \leq 0 \), but \( \infty \) otherwise. Figure 2.9 extends Wolfe's trip count formula for cases in which \( \alpha \) or \( J \) are unknown at compile time. However, the compiler must still insert an expression into the loop exit which appropriately reflects the various possibilities at run time.

From Figure 2.9, in cases (iii), (vii) and (ix), we have a situation where at run time \( \frac{-J}{\alpha} \) may be negative, but the trip count is \( \infty \). If we regard such a situation as either
programmer error or invalid input (since in scientific applications we do not anticipate infinite loops). we can safely ignore these cases. The other potential difficulty is that there may be a zero trip count at run time that cannot be detected at compile time. Note that if the quantity $\frac{-d}{a} < 0$. there is no problem with code generation. since the statement for($ix = 0; ix < \text{ceil}(-3/a); ix++$) will behave as if the trip count is 0. However, there are also cases where $\frac{-d}{a} > 0$ at run time. but the proper trip count is zero (this occurs in cases (iii) and (ix)). In these cases, the compiler can wrap the loop in a conditional test (a zero trip loop (ZTL) test) which only executes the loop if the trip count is nonzero at run time. regardless of the value of $\frac{-d}{a}$. This is a common way of
addressing the zero-trip loop problem [EHL91].

In cases where the trip count is obtained by symbolic division, a zero trip loop test must still be generated for the loop, since the TCTE (and hence, the trip count) may be zero even if the result of the symbolic division is positive. However, in this case, the compiler must use the original TCTE expression as the ZTL test, instead of the result of the symbolic division. An example of this can be seen in the loop in Figure 2.10.

Figure 2.10: ZTL Test

```c
N = foo();
i = 0;
ix = 0;
while(i < 10 * N)
{
    T = (10 * N) - i;
a[i] = 5;
i = i + N;
ix = ix + 1;
}
```

(a) loop before trip count analysis

```c
if((10 * N) - i > 0)
{
    for(ix = 0; ix < 10; ix++)
    {
        a[ix] = 5;
    }
}
```

(b) canonical loop with ZTL test

In (a), \( \alpha = -N \) and \( \beta = (10 \times N) \). so the symbolic division gives a trip count of 10. However, this is clearly only a valid value if the initial expression \( (10 \times N) \) is larger than zero. Thus, the compiler uses the TCTE expression \( ((10 \times N) - i) \) as a test at run time, instead of the constant 10.

When the trip count is computed using an explicit `ceil` call, the compiler must again insert a ZTL test, since the sign of the TCTE is unknown at compile time. In this case, the compiler can simply insert the resulting trip count expression itself into the enclosing `if` statement. The complete algorithm for trip count computation is summarized in Figure 2.11.
Figure 2.11: LCFN Trip Count Computation Algorithm

\[ \text{ivExpr} = \text{IV expression associated with TCTE} \]
if there is no IV expression
  mark L inadmissible
  return
ix = index variable associated with L
if ivExpr is not a linear function of ix
  mark L inadmissible
  return
compute \( \alpha, J \) such that ivExpr = \( \alpha \times ix + J \)
if \( \alpha \) and \( J \) are both literal constants
  compute trip count as \( \left\lceil \frac{-J}{\alpha} \right\rceil \)
if \( \alpha \) symbolically divides \(-J\)
  trip count = result of symbolic division
  mark L as requiring ZTL test
  return
if \( \alpha \) positive and \( J \) unknown
  trip count = 0
  return
if \( \alpha \) unknown and \( J \) negative or zero
  trip count = 0
  return
else
  construct trip count as \( \text{ceil}(-J / \alpha) \)
  mark L as requiring ZTL test

2.4 Subscript Normalization

Expressing array subscript expressions in a normalized form is also closely related to induction variable analysis. In order to determine if a given subscript can be expressed in the form \( a_0 + a_1 \times i_1 + \cdots + a_n \times i_n \), the compiler must classify the subscript as an induction expression in terms of the enclosing index variables \( i_1, \ldots, i_n \). An example of subscript normalization can be seen in the program in Figure 2.12.

In Figure 2.12(a) the three references to the array \( a \) all use the induction variable \( v \) in place of expressions involving loop index variables. In the normalized code in Figure 2.12(b), the appropriate induction expressions for \( v \) have been substituted in each case.
$i_1 = 0;$
$v = 0;$
while($i_1 < 10$)
{
    $a[v] = 0;$
    $i_2 = 0;$
    while($i_2 < 20$)
    {
        $i_3 = 0;$
        $a[v] = 5;$
        while($i_3 < 30$)
        {
            $a[v] = 5;$
            $v = v + 1;$
            $i_3 = i_3 + 1;$
        }
        $i_2 = i_2 + 1;$
    }
    $i_1 = i_1 + 1;$
}

(a)

for($i_3 = 0; i_3 < 10; i_3++]
{
    $a[600 \times i_3] = 0;$
    for($i_2 = 0; i_2 < 20; i_2++]
    {
        $a[30 \times i_2 + 600 \times i_3] = 5;$
        for($i_1 = 0; i_1 < 30; i_1++)
        {
            $a[i_1 + 30 \times i_2 + 600 \times i_3] = 5;$
        }
    }
}

(b)

2.5 Canonical Loop Generation

Generating a canonical loop is a fairly straightforward task once the loop trip count has been computed. Since the loop is already in while form, the conversion to canonical for form can be done directly. If a ZTL test is necessary, an if clause with the appropriate test expression wraps the for loop. This is summarized in Figure 2.13.
Figure 2.13: Canonical loop generation

**Input:** while loop L after trip count analysis and addition of loop index variable and update.

**Output:** for loop in canonical form equivalent to L

```plaintext
w_stmt = while statement of L
ix_init = statement preceding w_stmt
ix_upd = last statement of while body
remove ix_upd from loop body
tc = computed trip count for loop
create for statement with ix_init, ix_upd and tc
if L requires ZTL test
    ztl = ZTL test condition
    enclose for in if(ztl)
remove ix_init
replace while statement by generated for statement
```
Chapter 3

Pointer Array Access Normalization

In addition to the problems relating to the normalization of loop control flow, the C programming language also presents problems for dependence analysis because of the use of pointer variables by programmers. There are several ways in which such complications can occur:

(i) Use of pointer variables to reference scalars within a loop.

A pointer dereference that affects a scalar variable within a loop can obscure the effect of an array reference or a loop control variable from the compiler. This occurs in the loop in Figure 3.1.

Figure 3.1: Modification of scalars by pointer

```c
kz = 1;  // Start with a scalar
kz = *p;  // Reference to a pointer
kz = kx++;  // Pointer dereference
i = *i;  // Pointer dereference
p = *p - 1;  // Pointer modification
kz[0] = 5;  // Array reference
kz = kx + 1;  // Pointer dereference
```
If the compiler is unable to determine that the statement \( *p = *p + 1 \) updates the variable \( i \) by 1, it will be unable to determine that the array reference \( a[i] \) is equivalent to \( a[3*i+1] \). Furthermore, if the compiler cannot determine what variable the dereference of the variable \( p \) affects, it will have to make the conservative assumption that the array reference \( a[i] \) could refer to any array element, significantly reducing the efficacy of any dependence analysis. This type of difficulty arises in nearly any type of compiler analysis when there are unanalyzed pointer references and where conservative assumptions must be made in the absence of specific information. This affects such analyses as constant propagation, induction variable analysis, etc.

(ii) References made through dynamically allocated data structures. References to dynamically allocated data structures are generally made in C through pointers, and through explicit calls to the `malloc` library. These types of situations typically involve complex data structures such as linked lists and trees, which are difficult to analyze in any case, but ordinary arrays can also be used in this manner.

(iii) References to statically declared arrays made through pointer variables. C allows the equivalence of array references using explicit array indices, and equivalently using pointer dereferences. This is illustrated in Figure 3.2.

In this chapter, we will focus on dealing with the sorts of programs described by (iii). The primary difficulty with such programs is that the index expression(s) used to reference the array are implicit, and are hidden from the compiler because of the use of pointer arithmetic. For example in Figure 3.2(a), the index expression \( 2 * i \) is made implicit by the assignment of \( p \) before the loop, and by the increment to \( p \) which occurs on each iteration of the loop. Our goal is to recover such index expressions where possible, again expressing them in terms of loop index variables.

These types of problems are extremely difficult to deal with in the general case, because of the necessity for complex alias analysis when arbitrary pointer operations are allowed. Alias analysis is a problem that has been extensively studied. [HHN94, WL95, EGH94, JM81, LR92, Ban79, Bar77] but for which research is still progressing. Because of
Figure 3.2: Static array references via pointers for one and multi-dimensional arrays

<table>
<thead>
<tr>
<th>(a) One-dimensional array</th>
<th>(b) Multidimensional array</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[100];</td>
<td>int a[10][10];</td>
</tr>
<tr>
<td>int *p;</td>
<td>int *p;</td>
</tr>
<tr>
<td>p = &amp;a[0];</td>
<td>i = 0;</td>
</tr>
<tr>
<td>i = 0;</td>
<td>while(i &lt; 10)</td>
</tr>
<tr>
<td>while(i &lt; N)</td>
<td>{</td>
</tr>
<tr>
<td>{</td>
<td>p = a[i];</td>
</tr>
<tr>
<td><em>p = 5; /</em> a[2*i]=5; */</td>
<td>j = 0;</td>
</tr>
<tr>
<td>p = p + 2;</td>
<td>while(j &lt; 5)</td>
</tr>
<tr>
<td>i = i + 1;</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td><em>p = 5; /</em> a[i][2*j] = 5; */</td>
</tr>
<tr>
<td></td>
<td>p = p + 2;</td>
</tr>
<tr>
<td></td>
<td>j = j + 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>i = i + 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

the complexity of the problem, we will attempt to avoid alias analysis wherever possible, but will still attempt to deal with a reasonable range of programs.

3.1 PAN Algorithm

The aforementioned normalization, which we will term pointer array access normalization (PAN), operates on statically declared arrays. In addition, we make the following simplifying assumptions concerning the type of pointer operations that may occur in input programs:

(a) Any pointers in the program point only to statically declared arrays. There are no pointers to dynamically allocated data structures, nor are there pointers to scalars.

(b) For a given pointer variable p, any assignments to p in the program are of the form:

- \( p = \&(a[\varepsilon_1] \cdots [\varepsilon_k]) \)
where \( a \) is a statically declared \( k \)-dimensional array, and \( \varepsilon_1 \cdots \varepsilon_k \) are arbitrary expressions of integer type \(^1\), or

- \( p = a[\varepsilon_1] \cdots [\varepsilon_{k-1}] \)
  where \( a \varepsilon_1 \cdots \varepsilon_{k-1} \) are as above\(^2\)

- \( p = p \pm \varepsilon \)
  where \( \varepsilon \) is an arbitrary expression of integer type.

(c) A pointer variable \( p \) can only point to one array \( a \) during the course of the program, although it can be assigned multiple times using either form of assignment statement.

(d) Any pointer dereference in the program is of the form \( \ast p \) or \( \ast(p \pm \varepsilon) \), with \( p \) a pointer variable and \( \varepsilon \) an arbitrary expression of integer type\(^3\).

These simplifications allow the compiler to analyze array references without the need for sophisticated alias analysis in determining pointer-array relationships. Under these assumptions, determining which array \( a \) a given pointer points to requires only a simple scan of the program, and the relationship between a pointer and its associated array does not change during the execution of the program. Thus, the compiler can focus on converting pointer dereferences to equivalent array accesses based on the implicit access pattern created by whatever pointer operations exist.

In order to derive these index expressions, the primary idea employed is the observation that in programs such as the one in Figure 3.2(a) the assignments to pointer variables resemble the pattern of simple induction variables. This is a result both of the simplifying assumptions made and the fact that the C language restricts the manner in which pointer variables can be assigned. In Figure 3.2, we can consider the pointer \( p \) to be an induction variable of a special type, which is initially set to an offset of zero from the beginning of the array \( a \), and has its offset increased by a value of 2 on each loop.

\(^1\)Note also that \( \varepsilon \) should not contain side effects.
\(^2\)Note that for a one-dimensional array, this will be a simple assignment of the form \( p = a \)
\(^3\)In all of these cases \( \varepsilon \) should also be free of side effects.
iteration. This is exactly analogous to an ordinary integer induction variable which is assigned a value of zero before the loop and is incremented by 2 on each iteration.

3.1.1 One-Dimensional PAN

In this section, a PAN algorithm will be described for one-dimensional arrays which are accessed by pointer. In Section 3.1.2 PAN will be extended to handle multidimensional arrays. Assuming that the same type of induction variable analysis is available as was used in the LCFN algorithm (see Chapter 2), we can use it to do PAN by adding compiler-generated integer variables to the loop which correspond to the modifications of a pointer induction variable. The algorithm for this is summarized in Figure 3.3.

Figure 3.3: One-Dimensional PAN Algorithm

1. for each statement $S$ in $P$
2.  if $S$ is a pointer assignment $p = \& (a[\varepsilon])$
3.    if variable $I_{pa}$ does not already exist
4.     add dummy var $I_{pa}$ to symbol table
        \{$I_{pa}$ denotes var $I$ indexing array $a$ via pointer $p$\}
5.     add assignment $I_{pa} = \varepsilon$ immediately following $S$
6.  if $S$ is a pointer assignment $p = a$
7.     add assignment $I_{pa} = 0$ immediately following $S$
8.  if $S$ is pointer assignment of form $p = p \pm \varepsilon$
9.     add assignment $I_{pa} = I_{pa} \pm \varepsilon$ immediately following $S$
10. if $S$ contains a pointer dereference $*p$
11.    add assignment $R_{pa} = I_{pa}$ immediately before $S$
12. if $S$ contains a pointer dereference $*(p \pm \varepsilon)$
13.    add assignment $R_{pa} = I_{pa} \pm \varepsilon$ immediately before $S$
14. end for
15. run IV analysis
16. for each statement $S$
17.    if $R_{pa}$ is an induction variable at $S$
18.      let $\varepsilon = \text{induction expression associated with } R_{pa}$
19.      replace $*p$ in $S$ by $a[\varepsilon]$
20. end for

Essentially, this algorithm works by treating the pointer variable $p$ as an induction
variable. In the example illustrated in Figure 3.4, p is initialized to point to the first element of the array a before the loop, and is updated by a constant amount (via pointer arithmetic) on each loop iteration. Thus, on each iteration of the loop, the pointer p points to an element of a whose index value is an induction variable of the loop. The dummy variables \( I_{pa} \) and \( R_{pa} \) which are added into the loop by the compiler correspond to the index values resulting from pointer operations. The variable \( I_{pa} \) is used to model the effects of pointer assignments, while the variable \( R_{pa} \) is used to represent the value of the implicit index at program points where references to p are made. Associated with each type of pointer assignment is a corresponding assignment to \( I_{pa} \). A direct assignment to an array element using the & operator results in \( I_{pa} \) being assigned the corresponding expression. A pointer increment or decrement results in \( I_{pa} \) being incremented or decremented by a corresponding amount, respectively. In the case of a pointer dereference, \( R_{pa} \) is assigned the value of \( I_{pa} \) at that point in the program, unless the dereference also contains pointer arithmetic, in which case the additional increment or decrement is included in the assignment to \( R_{pa} \). Once these dummy variables are in place, p can be analyzed with ordinary IV analysis techniques. Corresponding pointer dereferences can then be converted to array accesses.

### 3.1.2 Multidimensional PAN

It is fairly straightforward to extend steps 1-15 in Figure 3.3 for multidimensional arrays referenced using pointers. However, even if the compiler has determined an induction expression for a given pointer reference in step 19, there can be difficulties in generating the appropriate array indices. This can be illustrated in loops such as those in Figure 3.5.

In Figure 3.5(a), we have a 2-dimensional array which is referenced within two enclosing loops. The proper array reference can be easily seen to be \( a[i][2 \times j] \) by inspection. However, it is more difficult in general for the compiler to derive the proper expression for each dimension. Specifically, the compiler must, given enclosing loop index variables \( ix_1, \ldots, ix_k \), an induction expression of the form \( J_1 \times ix_1 + \cdots + J_t \times ix_t \) (where \( J_t \) are loop invariant expressions), and a k-dimensional array \( a[u_1] \cdots[u_k] \), generate index expressions \( \xi_1, \ldots, \xi_k \) such that
Figure 3.4: PAN Processing Example

<table>
<thead>
<tr>
<th>source program</th>
<th>before IV analysis</th>
<th>final program</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[100];</td>
<td>int a[100];</td>
<td>int a[100];</td>
</tr>
<tr>
<td>int *p;</td>
<td>int *p;</td>
<td>int *ix;</td>
</tr>
<tr>
<td>i = 0;</td>
<td>i = 0;</td>
<td>for ix = 0;</td>
</tr>
<tr>
<td>p = &amp;a[i];</td>
<td>p = &amp;a[i];</td>
<td>ix = N - 1;</td>
</tr>
<tr>
<td>while(i &lt;= N)</td>
<td>while(i &lt;= N)</td>
<td>ix--;</td>
</tr>
<tr>
<td>*p = 5;</td>
<td>*p = 5;</td>
<td>a[2 * ix] = 6;</td>
</tr>
<tr>
<td>p = p + 2;</td>
<td>p = p + 2;</td>
<td></td>
</tr>
<tr>
<td>i = 1;</td>
<td>i = 1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\((A_1 \cdot \varepsilon_1 + \cdots + A_k \cdot \varepsilon_k = J_1 \cdot ix_1 + \cdots + J_l \cdot ix_l)\) subject to \(0 \leq \varepsilon_i \leq u_i\). and \(A_i = \prod_{j=i+1}^{k} u_j\). It is nontrivial for the compiler to derive these expressions, particularly if the expressions \(J_i\) are symbolic.

In Figure 3.5(b), a two-dimensional array is referenced with only a single enclosing loop. In a case such as this, even if the compiler could derive the proper index expressions, these expressions would necessarily involve \text{div} and \text{mod} operators. These types of expressions typically cannot be analyzed by dependence analysis techniques in any case.

In order to avoid these difficulties and handle multidimensional array references in a unified manner, \textit{array linearization} [BC86], [WB87] can be used to convert multidimensional arrays into equivalent one-dimensional arrays which can then be analyzed using the techniques of the previous section. Array linearization, which is summarized in Figure 3.6, has been used as a technique for doing dependence analysis on multidimensional arrays. There are some advantages and disadvantages to doing so, which are discussed by Girkar and Polychronopoulos [GP88].

\footnote{Since C requires the bounds of static arrays to be literal constants, \(A_i\) can be computed at compile time.}
Figure 3.5: Resolving multidimensional array references

```
int a[10][10];
int *p;

i = 0;
while(i < 10)
{
    p = a[i];
    Ipa = 10 * i;
    while(j < 5)
    {
        *p = 5;
        /* a[i][2 * j] = 5 */
        Rpa = Ipa;
        /* Rpa = (10*i) + (2*j) */
        p = p + 2;
        Ipa = Ipa + 2;
        j = j + 1;
    }
    i = i + 1;
}

(a)
```

```
int a[10][10];
int *p;

i = 0;
p = &(a[0][0]);
Ipa = 0;
while(i < 100)
{
    *p = 5;
    /* a[i/100][i % 100] = 5 */
    Ipa = Ipa + 1;
    i = i + 1;
}

(b)
```

The full PAN normalization algorithm can be summarized in Figure 3.7.
Figure 3.6: Array linearization

Given an array declaration \( a[l_1] \cdots [l_k] \):

1. \textbf{for} \( i = 1 \ldots k \)
2. \hspace{1em} compute \( A_i = \prod_{j=i+1}^{k} l_j \)
3. \textbf{end for}
4. compute \( S = \sum_{j=1}^{k} A_j \)
5. replace array declaration by \( a[S] \)
6. \textbf{for} each expression \( E \) in the program
7. \hspace{1em} if \( E \) is a reference \( a[\varepsilon_1] \cdots [\varepsilon_k] \)
8. \hspace{2em} compute expression \( E_1 = (A_1 \ast \varepsilon_1) + \cdots + (A_k \ast \varepsilon_k) \)
9. \hspace{2em} replace \( E \) by \( a[E_1] \)
10. \textbf{end for}

Figure 3.7: PAN Algorithm

1. \textbf{for} each array \( a \) in the program
2. \hspace{1em} if \( a \) is referenced by pointer
3. \hspace{2em} linearize(a)
4. \textbf{end for}
5. \textbf{for} each statement \( S \) in the program
6. \hspace{1em} if \( S \) is a pointer assignment
7. \hspace{2em} add appropriate index statement \( \text{(see Algorithm 3.3)} \)
8. \textbf{end for}
9. \textbf{for} each expression \( E \) in the program
10. \hspace{1em} if \( E \) is a pointer dereference
11. \hspace{2em} replace \( E \) by equivalent array reference \( \text{(see Algorithm 3.3)} \)
12. \textbf{end for}
13. run IV analysis
Chapter 4

Prototype Implementation

4.1 The Parafrase-2 Compiler

A prototype implementation of the LCFN and PAN methods described in the previous chapters has been built using the Parafrase-2 compiler environment. Parafrase-2 was chosen as a platform for several reasons. Firstly, Parafrase-2 is capable of compiling C programs and has a high-level intermediate representation which allows source-to-source transformations. Most importantly, it has an infrastructure which is extremely well suited to enabling admissible loop normalization. A number of supporting analyses which are necessary for admissible loop normalization, including constant propagation, subscript normalization, symbolic analysis, and especially induction variable (IV) detection, are present.

In addition to the passes which are part of the native compiler, Parafrase-2 allows additional passes to be added by the programmer. Each pass manipulates the intermediate form to implement code transformations. The PAN code has been implemented as a separate pass in the Parafrase-2 environment, and the code to implement LCFN has been attached directly to the induction elimination pass of Parafrase-2.

4.1.1 Overview of Parafrase

The primary benefit in using Parafrase-2 as a environment is the strength of its symbolic analysis framework [HP96]. Because of the strongly unified nature of this framework,
which is based on an abstract interpretation approach [CC77, CC79]. Parafrase-2 is able to implement supporting analyses in the presence of various syntactic structures, as well as symbolic expressions. The induction elimination and constant propagation passes are based upon this framework. The relationships among these native Parafrase-2 passes, and the added passes and code are illustrated in the following figure:

The symbolic interpretation engine represents the values of source expressions at compile time as multivariate polynomials of program variables in a canonical sum-of-products form. These abstract symbolic values are central to the computation of induction expressions, and allow the compiler to automatically generate normalized array indices without the need for further processing. This canonical representation also greatly facilitates other aspects of the implementation. For example, determining that a given induction expression is linear in a certain variable and then computing the expressions corresponding to the slope and intercept of the linear function is simple once the expression is in a unique canonical form.
Induction Expression Detection in Parafrase-2

The most important factor in successfully detecting admissible loops is the ability to detect induction expressions in the program. This relates directly to the ability to compute trip counts, as well as normalizing array index expressions, which are the primary factors in determining admissibility. Although many compilers implement analyses such as constant propagation and induction variable detection, Parafrase-2 offers several important features which allow a wider range of admissible loops to be detected:

- Induction expressions are represented in a canonical, program-point specific form, and are computed as explicit functions of a loop index variable. This is in contrast to simple IV detection algorithms, like that described by Aho, Sethi, and Ullman [ASU86], which express induction variables in terms of other program variables, but not necessarily in terms of a single index variable which expresses the iteration number of the loop. Also, simple algorithms do not take into account the fact that an induction variable may be modified more than once in a loop, and may have different characteristic functions\(^1\) at different program points. Furthermore, Parafrase-2 associates an induction expression with each source expression, rather than simply with program variables.

- Induction expressions can be recognized regardless of the syntactic form of the updates to induction variables. Some IV detection algorithms operate by scanning source code or pattern matching certain types of syntactic forms. Parafrase-2 computes IVs based on the semantics of program statements, not their syntax, abstracting away these differences, and thus is capable of recognizing a wider range of induction expressions.

- Induction expressions which result from updates along different control flow paths can be detected. This allows updates to variables along conditional control flow paths to be handled, and also allows updates to variables via inner loops to be detected.

---

\(^1\)The characteristic function of a variable is its closed form at a specific point in the source code.
modeled. This is valuable in practice, since we are often not dealing with single loops, but loop nests.

- Parafrase-2 is able to detect induction variables which are nonlinear polynomial or exponential functions of the loop index variable.

Enabling of IV Detection in C

Although the native Parafrase-2 compiler handles C, the induction elimination module was not fully operational for C programs. Hence, several fixes and additions were made to enable the module for use in admissible loop normalization:

- Code was added to create an explicit index variable for C while loops. In the case of a while loop, the new index variable is always initialized immediately before the while statement, its increment is always 1, and the increment occurs as the last statement in the while loop.

- Code was added to the induction module to reflect the addition of new variable introduced above, and modify the appropriate internal data structures in the module. This ensures that the symbolic execution engine will correctly execute the modified loop, since the source program being compiled is being changed.

- Parafrase-2 represents the trip count of FORTRAN DO loops during induction variable analysis, and the symbolic analysis engine uses the values to model the entire effect of a loop on program variables. This also enables the detection of multiloop induction variables. Code was added to the phase of the induction module in order to return an appropriate abstract symbolic expression to the IV analysis for while loops. The native IV module extracts a loop count directly for FORTRAN DO loops, but does not do so for any other type of loop. If the trip count computed by the admissible loop analysis is a constant, then an equivalent abstract symbolic constant is returned. If the computed trip count is the result of a symbolic division, then again an equivalent abstract symbolic expression is returned to the induction
module. If the trip count involves a call to the C `ceil` function. a NULL expression is returned (indicating an unknown trip count) ².

Since much of the existing Parafrase-2 code is not as well enabled for C as for FORTRAN. several other fixes were made to the existing code to allow C to be handled properly. This process was hindered somewhat by the lack of detailed documentation available on the implementation of the native Parafrase-2 passes. In addition, the Parafrase-2 infrastructure does not prevent the programmer from making inconsistent or invalid modifications to the syntax tree. This. combined with the lack of native routines to do some common types of manipulations of source code constructs. and sparse documentation of existing routines. also hindered implementation somewhat.

4.1.2 LCFN Implementation

The code to implement LCFN has been added directly to the induction elimination module. Since this module handles induction variable analysis and subscript normalization automatically. the added code implements the preprocessing necessary to generate canonical loops. and interrupts the induction variable analysis to generate the necessary information for canonical loop generation as each loop is processed. Once the induction elimination module is finished processing. any canonical loops generated are then inserted into the code.

4.1.3 PAN Implementation

Code to implement array access normalization has also been implemented in Parafrase-2. All of the code necessary for the purposes of PAN implementation has been written as a prepass which makes the appropriate changes to the input program before the invocation of the Parafrase-2 induction pass. Since Parafrase-2 automatically computes normalized subscripts for any array accesses in the loop. the PAN prepass first linearizes arrays where necessary. and then converts any pointer deferences to their equivalent array forms.

²The symbolic analysis engine of Parafrase-2 can only represent values that are polynomial functions of program variables. However, \([p(x)]\), for \(p(x)\) a polynomial. cannot itself be represented by a polynomial.
1. Run FORTRAN 77 code through Parafrase-2 dependence analysis
2. Generate statistics on parallelizable loops for FORTRAN code
3. Convert FORTRAN 77 code to C
4. Modify C code to obscure loop accesses from dependence analysis
5. Run admissible loop transformation on C code
6. Apply dead code elimination/fixes to C code
7. Run normalized C code through dependence analysis and generate statistics

regardless of whether the array index expression is an induction expression or not. If so, the Parafrase-2 induction pass automatically completes the PAN process by substituting appropriate induction expressions (if any) for the array index. If not, the array access is left with an unnormalized index expression.

4.2 Experimental Evaluation

In order to evaluate the prototype implementation, three benchmark applications were chosen as sample inputs for the admissible loop transformation. These benchmarks are the *tomcatv* program contained in the SPEC95 benchmark suite and the *embar* and *conjugate gradient* applications which are part of the Numerical Aerodynamics Simulation (NAS) parallel benchmark suite [BB94].

4.2.1 Experimental Procedure

The procedure used to evaluate the implementation is summarized in Figure 4.1. Since the aforementioned applications are coded in FORTRAN 77, each was converted to C using the GNU f2c FORTRAN-to-C translation tool [FG90]. f2c provided a base of C code which was then modified by hand in order to obtain code which obeys the syntactic requirements of the Parafrase-2 implementation. In most cases, only minimal changes were required. Additional modifications were made to the program loops in order to introduce difficulties for the dependence analyzer. These modifications included:

(a) conversion of f2c-generated *for* loops to *while* loops to obscure index variables and
loop trip counts from the compiler.

(b) replacement of array indices by expressions involving introduced loop induction variables.

(c) replacement of array access expressions by equivalent pointer dereference expressions.

(d) use of varying loop exit conditions.

Changes made to the code to conform to syntactic constraints included 3:

(a) f2c-generated goto statements replaced by appropriate structured constructs.

(b) f2c-generated << operators replaced by multiplications.

(c) f2c-generated ++, +=, etc operators by equivalent explicit operators4.

(d) explicit code introduced corresponding to FORTRAN MAX and ABS functions5.

Dead code elimination and the insertion of declarations for variables introduced by the induction module were accomplished by hand after the admissible loop normalization phase because of difficulties encountered with the Parafrase-2 implementation for C.

The data dependence pass of Parafrase-2 provides a dependence analyzer for array references which have simple linear index expressions. Parafrase-2 uses the gcd and bounds dependence tests to construct a data dependence graph (DDG). Parafrase also provides the dotodoall pass, which analyzes the DDG in order to mark each DO loop as parallel or non-parallel, based on the dependence information generated. These passes were used to generate a count of parallelizable loops for each application. Although the data dependence and dotodoall passes were only partially enabled for C6, an extra mini-pass was written which modified the internal C syntax tree of each admissible for loop

---

3 f2c introduces scalar pointer variables to represent reference parameters in function calls. This technically violates the requirements of admissibility/IV analysis. The Parafrase-2 modules do not analyze C pointer syntax, and since none of the introduced code affects either the dependence analysis or the IV analysis, this does not affect the final results.

4 The Parafrase-2 symbolic analysis engine does not support these C operators.

5 Although the presence of certain calls in FORTRAN (i.e. SQRT, LOG) does not present problems for Parafrase-2, the equivalent calls did so in C. For this reason, these calls were temporarily removed during the analysis to allow the appropriate IV analysis to proceed.

6 In particular, they analyzed individual statements properly, but only handled FORTRAN DO loops.
Figure 4.2: Parallel Loop Statistics

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Loops</th>
<th>Parallel Loops</th>
<th>Admissible Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>embar (NAS)</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cg (NAS)</td>
<td>31</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>tomcatv (SPEC95)</td>
<td>16</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

to correspond to that of a FORTRAN DO. This allowed dependence and parallelizations results to be generated for the normalized C' code as well.

4.2.2 Experimental Results

Parallelization Results

Figure 4.2 summarizes the results obtained for the three benchmark applications.

As can be seen from the figure, exactly the same parallel loops were detected for each application in FORTRAN and in C, indicating that the dependence analyzer was able to process the loops and determine that the loops were parallelizable, despite the presence of the problematic C' constructs. Thus, dependence analysis for C' was enabled.

Dependence Results

The ability of the admissible loop normalization to enable dependence analysis for C' can also be illustrated by examining selected loops from the benchmark applications in further detail.

The following loop from the embar application illustrates a parallelizable loop that has no loop-carried dependences. The loop is a simple initialization of an array, as seen in Figure 4.3.

Although in FORTRAN this loop can clearly be parallelized, in C the dependence analyzer must report dependence for the loop if the assignment \( *p = 0 \): cannot be analyzed\(^7\). However, in the normalized loop, the syntax matches that of the FORTRAN

---

\(^7\)Note however that the C dependence analysis is not properly enabled in the native Parafrase-2 code and it will incorrectly ignore the possible effect of the pointer dereference.
Figure 4.3: Sample loop from embar application

<table>
<thead>
<tr>
<th>FORTRAN loop</th>
<th>unnormalized C loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 110 I = 0, NQ - 1</td>
<td>i__ = 0;</td>
</tr>
<tr>
<td>Q(I) = 0.d0</td>
<td>p = &amp;q[0]);</td>
</tr>
<tr>
<td>110 CONTINUE</td>
<td>while(nq - 1 &gt;= i__)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>*p = 0;</td>
</tr>
<tr>
<td></td>
<td>p = p + 1;</td>
</tr>
<tr>
<td></td>
<td>i__ = i__ + 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

(a) FORTRAN loop              (b) unnormalized C loop

<table>
<thead>
<tr>
<th>normalized C loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(ix = 0; ix &lt; nq - 1; ix = ix + 1)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>a[ix] = 0.;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

(c) normalized C loop

loop and the loop is detected as parallelizable.

We can also look at an example of a loop which does carry dependences and thus cannot be parallelized. Such a loop can be found in the tomcatv application (see Figure 4.4).

In this case, the inner I loop has no loop-carried dependences. This loop can be parallelized. However, the outer J loop has a flow and anti dependence carried between the assignment and references to the arrays RX and RY respectively. Each of the four resulting dependences have a dependence distance of 1. Thus, the outer loop cannot be parallelized. In the unnormalized C loop, the array references and the loop trip counts have been obscured by use of the while loops, as well as by the introduction of induction variables v1, v1, and v2 to the array index expressions. In the normalized C code generated by the admissible loop normalization, these array references have been converted back to expressions involving loop index variables. The Parafrase-2 dependence analyzer

---

8Also, the array d__ has been linearized. This is a result of transformations elsewhere in the program.
detects the same four dependences for the normalized C loop, and correctly determines that the inner loop is parallelizable. The dependences detected for the tomatcv application are summarized in Figure 4.5 . Each of the 16 loops in the program are listed, with the count of dependences detected for each. Note that the parallelizable loops are those which have a dependence count of zero.

Although the dependences detected match exactly for those loops which are paral-

---

9Loops L1, L2 and L16 are I/O loops and were not coded in C. Loop L3 was inadmissible in C and thus not analyzed for dependence.
Figure 4.5: Dependences for tomcatv

<table>
<thead>
<tr>
<th>Loop</th>
<th>Dependencies</th>
<th>Admissible</th>
<th>Loop</th>
<th>Dependencies</th>
<th>Admissible</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1</td>
<td>-</td>
<td>L9</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>L2</td>
<td>1</td>
<td>-</td>
<td>L10</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>L3</td>
<td>296</td>
<td>-</td>
<td>L12</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>L4</td>
<td>59</td>
<td>60</td>
<td>L13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L5</td>
<td>59</td>
<td>60</td>
<td>L14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L6</td>
<td>2</td>
<td>30</td>
<td>L15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L7</td>
<td>2</td>
<td>30</td>
<td>L16</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>L8</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In loops L4.L5.L6.L7.L9.L10 and L12, there are differences in the dependences detected between the FORTRAN code and the corresponding C code. In the case of L6 and L7, this is due to differences in the coding of these programs. In particular, the C versions of these loops have explicit code to compute the FORTRAN MAX and ABS functions, leading to multiple dependences which correspond to simple function calls in FORTRAN. The other loops require further analysis and are shown in Figure 4.6 (loops L12 and L13 are shown in Figure 4.4).

In loops L4 and L5, there is a single extra output dependence detected by the Parafrase-2 dependence analyzer because of the linearization of the array dd, which in Figure 4.6(a) is assigned by the two-dimensional reference DD(I,J), but in C has been converted to dd[514*ix1+ix2+1030]. Although there is no actual dependence, the Parafrase-2 dependence analysis is relatively unsophisticated and cannot analyze the linearized reference. Because both loop index variables appear in C within a single array index expression, Parafrase-2 treats the other index variable within each loop as an unanalyzed symbolic variable within the array index, and thus reports dependence. A similar problem occurs with loop L12. In this case, the original FORTRAN loop counts downwards with an increment value of -1, but the normalized C code has converted the loop to count upwards with an increment value of 1. As a result, the array references RX(I,J) and RY(I,J) become rx[ix9+2][-ix10+n-2] and ry[ix9+2][-ix10+n-2] respectively. The variable n then becomes an extra symbolic value in the array reference.
leading to an extra dependence.

In loop L10, there are an extra output, flow and anti dependence detected because of the reference and assignment to the linearized array d_ in the loop. Similarly, in loop L9, there is an extra output dependence because of the assignment to d_. However, in the case of L9, the total number of dependences detected is actually smaller because two invalid dependences (antidependences with dependence distance -1) which are erroneously detected by Parafrase-2 in FORTRAN do not appear in C.

Figure 4.6: Non-parallelizable loops in tocorp

<table>
<thead>
<tr>
<th>(a) L4, L5 (F77)</th>
<th>(b) L4, L5 (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 40 J = 2: N+1</td>
<td></td>
</tr>
<tr>
<td>DO 50 I = 2: N+1</td>
<td></td>
</tr>
<tr>
<td>E = B<em>B</em>ATXEL</td>
<td></td>
</tr>
<tr>
<td>CONTINUE</td>
<td></td>
</tr>
<tr>
<td>CONTINUE</td>
<td></td>
</tr>
<tr>
<td>FOR IX2=0; IX2 = n-2; IX2--</td>
<td></td>
</tr>
<tr>
<td>IF: n-1 - 1 = 0</td>
<td></td>
</tr>
<tr>
<td>FOR IX1=1; IX1 = n-2; IX1--</td>
<td></td>
</tr>
<tr>
<td>E = E - a^* REL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c) L9, L10 (F77)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 10 J = 1: N+1</td>
</tr>
<tr>
<td>DO 10 I = 1: N+1</td>
</tr>
<tr>
<td>BK = AA I, J</td>
</tr>
<tr>
<td>BK = BK*AA J, I</td>
</tr>
<tr>
<td>RX I, J = RX I, J + RX I, J 2</td>
</tr>
<tr>
<td>RY I, J = RY I, J - RY I, J 2</td>
</tr>
<tr>
<td>CONTINUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(d) L9, L10 (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR IX7=1; IX7 = n-2; IX7--</td>
</tr>
<tr>
<td>IF: n-1 + 1 = 1</td>
</tr>
<tr>
<td>FOR IX5=1; IX5 = n-2; IX5--</td>
</tr>
<tr>
<td>3 = AA I, IX5 2</td>
</tr>
<tr>
<td>d__ = (IK14*IX5-IX7-1.31) = 1</td>
</tr>
<tr>
<td>3 = (IK14*IX5-IX7-1.31)</td>
</tr>
<tr>
<td>RX I, IX5 = RX I, IX5 2</td>
</tr>
<tr>
<td>3 = (RX I, IX5-IX7-1)</td>
</tr>
<tr>
<td>RY I, IX5 = RY I, IX5 2</td>
</tr>
<tr>
<td>3 = (RY I, IX5-IX7-1)</td>
</tr>
</tbody>
</table>
Chapter 5

Related Work

The problem of supporting dependence testing in C has also been tackled by Justiani and Hendren [JH94]. They have implemented support phases to convert some types of C loops to a canonical form similar to that implemented by LCFN, within the MCCAT compiler environment. The differences between the MCCAT analysis and that of LCFN can be summarized as follows:

(a) MCCAT assumes that any loop is defined as a for loop with an explicit initialization, increment, and test of a single loop variable\(^1\). Thus, MCCAT does not handle missing or implicit index variables, nor does it compute loop trip counts.

(b) MCCAT does not allow the loop body to modify loop control variables. Such modifications are detected by MCCAT, but result in the loop being marked as inadmissible.

(c) MCCAT analyzes scalar pointer references, but still requires that array references be made using explicit index expressions. Array references made through pointers are not handled.

(d) MCCAT incorporates analysis of scalar pointer references into the induction variable detection and the subscript normalization process, supported by the strong points-to-analysis alias analysis available in the MCCAT compiler. Thus, MCCAT can

\(^1\)A scalar pointer variable of the form *\(p\) can be used as the loop index variable.
incorporate the effects of such dereferences into the IV analysis. LCFN and PAN do not deal with scalar pointers.

(e) MCCAT also can analyze some types of stack-based aliases between array names, so that an array which is accessed using a different name can be detected. PAN makes simplifying assumptions about the form of pointer assignments to make the relationship between pointer and array unambiguous.

The primary strength of the analyses implemented in MCCAT over those present in Parafrase-2 is its ability to handle scalar pointer references and incorporate their effects into the other analyses needed to implement admissible loop normalization. Conversely, LCFN and PAN handle a wider range of syntactic structures with respect to loop control flow, and can analyze some types of pointer-based array references, which MCCAT does not handle.

There are several research projects currently involved with developing extensions to the C++ programming language. for the purposes of parallel computing. The $p$C++ project [BBG91] and the $c$C++ project [CK93] have defined extensions to the C++ language for the purposes of providing a model for parallel C++ programming. In addition, the HPC++ project [BGJ95] has focused on providing a runtime library, as well as compiler directives to provide parallel programming support. HPC++ provides loop directives for the purposes of parallelizing well-behaved loops under conditions which are similar to those defined by a canonical loop. In particular, the $\text{HPC\_INDEPENDENT}$ directive allows the compiler to parallelize loops, provided that the loop:

(a) The loop is a $\text{for}$ statement.

(b) The loop termination condition involves only loop IVs.

(c) The loop update condition only modifies loop IVs.

The most important aspect of admissible loop normalization is $\text{induction variable analysis}$. IV analysis is needed to normalize array subscripts, compute loop trip counts, and resolve array indices for pointer-based array references. The simplest IV algorithms.
such as that described by Aho et al. [ASU86] detect variables whose only assignments within a loop are increments or decrements by a constant value. This allows such variables to be classified as linear functions of other IVs. However, since IV algorithms were initially used for the purposes of strength reduction, these are not necessarily expressed in terms of a loop index variable. Furthermore, the Aho et al. algorithm depends on detecting specific syntactic forms for updates, and does not deal with internal loop control flow.

Wolfe [Wol92] describes a more advanced IV algorithm designed specifically for the purposes of advanced loop transformations used in parallelizing compilers. Wolfe’s method uses an analysis based on the SSA form [CFR91] to find linear induction expressions in loop nests. Wolfe’s method is capable of detecting multiloop induction variables in which the initial value or step of the induction variable occurring in an inner loop may vary in an outer loop. Wolfe’s method is also capable of detecting other types of induction expressions, including wrap-around variables, periodic variables, and monotonic variables. Wolfe’s IV analysis is also used for the purpose of computing loop trip counts, and his method has been extended in this thesis to deal with cases in which symbolic expressions are present.

IV techniques like that of Wolfe, and that in the Polaris compiler [PE94], are also capable of detecting nonlinear induction variables, which are polynomial or geometric functions of the loop index. These types of IVs can arise in triangular loop nests, which appear in some scientific applications, or in situations in which an IV is updated by a nonconstant value on each loop iteration.

The IV analysis existing in the Parafrase-2 compiler is also capable of detecting nonlinear induction variables, and can detect multiloop IVs within loop nests. The Parafrase-2 IV detection is very strong, and as such provides an excellent framework on which to base admissible loop normalization. The Parafrase-2 IV analysis is based on a symbolic analysis framework, in which the source program is executed within an abstract domain representing the symbolic values of program variables at run time. Parafrase-2 is capable of representing expressions which are multivariate polynomials or exponentials. Parafrase-2 symbolically executes loops, and uses symbolic interpolation to attempt to fit the sequence of values assumed by a given expression to a polynomial or exponential
function. Because Parafrase's approach is based on symbolic execution as opposed to pattern-matching or ad hoc approaches, it detects IVs on a program-point specific basis. In addition, it handles multiple updates to IVs, equivalent updates along different control flow paths, and automatically models the effects of inner loops on IVs if the loop trip count is known.

The problem of accurate alias analysis for C programs is one to which increasing attention has been paid, but for which solutions amenable to practical use in real compiler systems have not yet been developed. Early work in detecting alias relationships in programs [Ban79, Bar77] focused on FORTRAN-like programming languages, in which the primary source of aliases is the use of reference parameters in procedure calls. Recent work has focused on attempting to handle the more complex types of pointer relationships that occur in C programs as a result of various C features. These include:

- the creation of new pointer relationships with the C & operator.
- multilevel pointer references (i.e. **p).
- interprocedural alias relationships, including those created by recursive functions.
- pointer analysis for dynamically allocated data structures, as well as statically allocated variables.
- the use of function pointers in C.
- the use of type casts and pointer arithmetic.

All of these aspects of C can cause significant complications for an alias analysis scheme. A basic problem occurs in handling dynamically allocated data structures because there are not explicit names for heap objects, as contrasted to statically allocated objects on the stack. In order to handle recursive data structures, some techniques [LR92, JM81] limit the depth to which recursion is modeled by \textit{k-limiting} to keep a finite number of object names. This, however, can lead to overly conservative information. Emami, et al. [EGH94] and Wilson and Lam [WL95] both describe schemes to do context-sensitive interprocedural alias analysis for C programs. Emami, et al. attempt
to separate stack-based and heap-based pointer relationships, and reanalyze a procedure for each of its calling contexts. This approach has exponential complexity in the worst case, however. Wilson and Lam attempt to avoid this problem by summarizing only those relationships between procedure parameters that actually occur in the program. They also analyze pointers at a low level to avoid problems caused by type casting and compound data structures. Hummel. et al. [HHN94] describe a scheme for analyzing complex pointer data structures such as trees and linked lists, but require information specifying relevant properties of the data structures being used.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

Existing dependence analysis techniques rely on the compiler's ability to construct appropriate dependence equations from the program source code. This requires a regular loop syntax, normalized array references, and a known loop trip count. Enabling dependence analysis in the C language relies on being able to transform as many of the diverse possible syntactic structures for loop control flow and array references that are available to the programmer in C into equivalent forms with which the dependence analyzer can effectively deal. The primary complications that arise in C result from the use of non-for loop syntax, implicit or missing loop control constructs, and the use of pointers to reference static arrays.

LCFN has been implemented in the Parafrase-2 parallelizing compiler to convert certain types of C loops into a canonical form obeying the basic assumptions of dependence analysis. LCFN uses induction variable analysis to compute a trip count for the loop and to compute normalized expressions for array access expressions within the loop, from which a canonical for loop can be generated.

PAN has also been implemented in Parafrase-2 to allow the compiler to handle simple forms of implicit array references inside loops via pointers. PAN also uses induction variable analysis to attempt to re-extract implicit array index expressions from array references. In the case of multidimensional arrays, PAN uses array linearization in order
to resolve index expressions in the presence of multiple dimensions and multiple enclosing index variables.

The LCFN and PAN implementations were tested on sample FORTRAN benchmarks extracted from the NAS and SPEC95 benchmark suites and converted to C. After loop normalization was applied, the Parafrase dependence analyzer was able to detect the same parallel loops in C as in the original FORTRAN programs, indicating that the dependence analysis process was successfully enabled for C. However, the linearization of arrays was found to cause problems for simple dependence analyzers which are not capable of handling symbolic terms in array reference expressions.

6.2 Future Work

There are several ways in which the range of loops which are detectable as admissible loops might be extended. Some of these are summarized as follows:

- Handling of scalar pointer references.

  The most obvious way to extend LCFN and PAN would be to merge those analyses with the type of scalar pointer alias analysis available in the MCCAT compiler. Both scalar and array pointers tend to be used fairly widely in the C language, and a compiler must be prepared to deal with both. Because Parafrase-2 does not have a C alias package natively which has the strength of the alias analysis present in MCCAT, scalar pointer analysis has not been incorporated into the Parafrase-2 implementation. However, the unified nature of Parafrase-2's symbolic execution engine would allow the supporting analyses such as constant propagation and IV elimination to be enabled by incorporating the effects of pointer operations into the symbolic execution engine.

- Computation of loop trip counts from nonlinear induction expressions.

  Given that many compilers are now able to detect nonlinear induction variables, it is reasonable to attempt to extend trip count computation for TCTEs which are polynomial functions of the loop index variable. As in the linear case, this is a
matter of finding the smallest value of the loop index variable such that the function $f$ representing the TCTE is nonpositive. Doing this for an arbitrary polynomial function is difficult, particularly if symbolic terms are involved in the function. However, it may be possible to do this in cases that arise in practice where $f$ is a quadratic or 3rd order polynomial and the roots of the polynomial can be computed analytically.

- Computation of loop trip counts from boolean exit conditions.

The current LCFN implementation is capable of handling loop exit expressions that are arithmetic comparisons. An additional extension to this would be to allow exit conditions which are boolean combinations of such arithmetic comparisons, using AND, OR and NOT operations. However, it is uncertain whether such exit conditions would occur often enough in real programs to be useful. In order to implement these types of operators in general, the compiler would need to represent ranges of iterations over which a given condition is true, since an AND operation requires that both of the component conditions be true simultaneously. Obtaining a loop trip count expression would involve executing the appropriate intersection or union operations on iteration ranges corresponding to each arithmetic comparison condition. These operations are easy to implement where the bounds of ranges are known constants, but are problematic where the bounds involve symbolic terms. However, Blume and Eigenmann [BE95] have implemented an extension of the range test in the Polaris compiler for computing symbolic ranges at compile time.
Appendix A

Induction Variable Analysis

Induction variable (IV) analysis is an important part of many optimizing and parallelizing compilers. Although the exact definition of an induction variable has not always been completely consistent and has tended to evolve along with the techniques for detecting them, a fairly general definition can be given as follows:

**Definition 3** Given a loop L and a variable v, the variable v is an induction variable in L if the sequence of values assumed by v at a given program point within the execution of the loop L can be represented by a function \( f(i) \), such that \( i \) represents the number of a particular iteration of L, and \( f \) is an analytic function of a certain type. Specifically, \( f \) may be of several different types:

(i) \( f \) is a linear function of \( i \), of the form

\[
f(i) = \alpha \times i + J \text{, where } \alpha \text{ and } J \text{ are compile time constants or invariant expressions in } L.
\]

(ii) \( f \) is a polynomial function of \( i \) of the form

\[
f(i) = a_0 + a_1 i^1 + \cdots + a_n i^n \text{ where } n \text{ is a compile time constant, and } a_0, \ldots, a_n \text{ are either constants or invariants in } L.
\]

(iii) \( f \) is an exponential function of the form

\[
f(i) = e^{g(i)} \text{ where } c \text{ is constant or invariable in } L, \text{ and } g(i) \text{ is a linear function of } i.
\]
Following the terminology given by Haghighat and Polychronopoulos [HP96], the function $f$ can be termed the characteristic function for the IV $v$. Note that a given IV may have different characteristic functions at different program points. These different types of induction variables are illustrated in Figure A.1. IVs can also have different characteristic functions with respect to different loops in a loop nest. The concept of an induction variable can also be extended to define an induction expression [HP96], which is any program expression whose value can be represented as above.

Linear induction variables often occur as a result of accessing arrays by step within a loop, and are illustrated in Figure A.1(a). Typically, a linear IV occurs as a result of an update by a constant or invariant expression on each loop iteration, but this can also occur as a result of linear combinations of other IVs. Figure A.1(d) also illustrates a linear IV that is updated through the effect of an inner loop, as opposed to an explicit assignment. Polynomial induction variables commonly occur in triangular loop nests, as illustrated in Figure A.1(c). Figure A.1(b) illustrates an exponential IV, which arises from a multiplication on each loop iteration instead of an addition.

Many similar but different definitions of induction variables have been presented in the literature. The definition presented by Haghighat and Polychronopoulos [HP96] is closest to that presented here, because it is a definition based on program semantics, as opposed to ones based on the syntactic forms of variable updates. This type of definition is desirable because it clearly differentiates between what an induction variable is from the methods used by a compiler to detect them. Most of the definitions of the latter kind define an induction variable as a variable whose assignments within the loop have a specific form, generally that of a increment statement $v = v + c$. The most basic IV definitions, such as that given by Aho, Sethi, and Ullman [ASU86] require $c$ to be a literal constant. Others, such as that given by Pottenger [Po95] allow increments by loop invariant values, as well as coupled inductions in which IVs may appear in the increment of other induction variables. Wolfe [Wol92] differentiates between basic induction variables, which are obtained through simple increments or coupled inductions, and other induction variables which are linear combinations of other IVs. Wolfe also defines multiloop induction variables, which are variables that occur in an inner loop, and whose initial
value is an induction variable in an outer loop, while being incremented in the inner loop by a value that is invariant in the outer loop.

Along with the various definitions of induction variables are different methods for detecting them. Approaches that define induction variables in terms of the syntax of the variable updates typically rely on the simplified form of these updates to derive the characteristic formula for the variable. For example, Aho et al. [ASU86] require that all updates to a variable by of the form $v = v + c$, where $c$ is a constant, although multiple updates are allowed. Under these simplifying assumptions, IVs can be detected via a simple scan of the source code, after loop invariant computation has been done. However, these type of approaches implicitly assume that the compiler can always determine immediately which variables are referenced by each statement, which is not always pos-
sible in real programs. More complex approaches use more complex supporting analyses to detect IVs while taking into account the effects of nested loops, as well as control flow within the loop. Wolfe [Wol92] uses an approach based on SSA form, to relate the detection of different types of induction variables to certain types of graph-theoretic problems. Pottenger [Po95] describes an algorithm which recursively models inner loops, while computing the effects of variable updates by additions or multiplications. The total computed effect on a variable in a single iteration is used to derive the closed form. Very advanced approaches, such as that described by Haghighat and Polychronopoulos [HP96] operate by executing the loop in an abstract domain (abstract interpretation) and deriving closed forms for induction variables from the sequence of values obtained on each simulated iteration. In particular, this approach uses Newton's interpolation formula to fit the sequence of values obtained for the variable to a polynomial or exponential function. This type of approach is very powerful because it depends only on the operations that can be symbolically modelled, not on specific syntactic forms.

Figure A.2: Induction variable elimination

<table>
<thead>
<tr>
<th>(a) loop with IV</th>
<th>(b) loop after IV elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>liv = 0;</td>
<td>for(ix = 0; ix &lt; N; ix++)</td>
</tr>
<tr>
<td>for(ix=0; ix &lt; N; ix++) {</td>
<td>{ a[3 * ix] = 0;</td>
</tr>
<tr>
<td>liv = liv + 3;</td>
<td>}</td>
</tr>
<tr>
<td>a[liv] = 0;</td>
<td>}</td>
</tr>
</tbody>
</table>

Induction variables are important in parallelizing compilers because their detection often allows the compiler to eliminate dependences within a loop through induction variable elimination, as illustrated in Figure A.2. In Figure A.2(a), the variable liv hinders parallelization, because the assignment statement causes an output dependence in the loop. However, in Figure A.2(b), in which liv has been eliminated and the array index replaced by the equivalent expression (3 * ix), this dependence does not exist. Note that early uses of IV analysis attempted to detect IVs for exactly the opposite reason: namely, to make loop execution more efficient for serial machines through strength re-
duction. In Figure A.2(a), the relatively expensive multiplication in each loop iteration has been replaced by a less expensive addition operation, so the code in Figure A.2(a) is actually preferable on a serial machine. For the purposes of admissible loop normalization, induction variable analysis is needed in several contexts. Firstly, the ability to detect linear induction variables is necessary in order to compute loop trip counts, using the techniques described by Wolfe [Wol92]. IV analysis is also important in subscript normalization, where array subscript expressions are rewritten in terms of enclosing loop index variables. In a similar way, IV analysis is needed in order to detect the array access patterns caused by pointer arithmetic operations.
Bibliography


