A PRACTICAL SYMBOLIC ARRAY DEPENDENCE ANALYSIS FRAMEWORK FOR C

by
Christopher Lapkowski

School of Computer Science
McGill University, Montreal

March 1997

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Abstract

Array dependence analysis is an important component of any optimizing/parallelizing compiler. The results of array dependence analysis are required by many parallelizing transformations as well as other analyses for further optimizations.

This thesis is concerned with a symbolic framework for array dependence analysis. It has been shown that the presence of unknown symbolic values in array index expressions occurs with significant frequency such that the accuracy of array dependence analysis is compromised. The use of helper analyses, such as constant propagation, resolves the problem in some situations, but in general the development of symbolic array dependence analysis is needed. This thesis shows how such a symbolic framework can be implemented for the complete programming language C, including accurate handling of multi-level pointers.

In addition to the major contributions to the development of a symbolic framework and how it applies to array dependence analysis, this thesis has made contributions in several other areas. First, an analysis called *Extended SSA Numbering* was developed that closely resembles *Static Single Assignment* form but accommodates multi-level pointers and provides additional information for pointer references. Next, a compact and effective representation for a symbolic expression was developed. This representation preserves intermediate and all possible values of the expression so that if it is compared with a similar expression the comparison results will be meaningful. Lastly, the ability to compare array references with symbolic index expressions permits the comparison of array references within the same basic block or same hierarchical block. This information is gathered by the analysis called *Hierarchical Array Dependence Analysis* which then helps in detecting parallel statements, and improves instruction scheduling by reducing the data dependency graph.

This work has been integrated into the McCAT optimizing/parallelizing C compiler and in particular into the McADT (the McCAT Array Dependence Tester). Based on this implementation, several experimental studies were conducted and the results of these experiments are discussed in detail.

Résumé

L'analyse de dépendance des tableaux est partie intégrante de tout compilateur optimisant. Les résultats de cette analyse sont requis par plusieurs transformations parallélisantes, ainsi que par d'autres analyses d'optimisation.

La présente thèse traite d'un cadre symbolique pour l'analyse de dépendance des tableaux. Il est prouvé que la fréquence de symboles dont la valeur est inconnue dans des expressions représentant les indices de tableaux compromet la précision de l'analyse de dépendance de tableaux traditionnelle. D'autres analyses, comme la propagation de constantes, résolvent le problème dans certains cas, mais généralement un développement de l'analyse de dépendance des tableaux est nécessaire. La présente thèse démontre l'implantation d'un cadre symbolique pour le langage de programmation C dans son intégralité, incluant le traitement adéquat de pointeurs multi-niveaux.

En plus de contribuer au développement d'un cadre symbolique et à son application à l'analyse de dépendance des tableaux, la présente thèse contribue aux trois domaines suivants. D'abord, une analyse intitulée Numérotation SSA Etendue est développée, qui est proche de l'analyse SSA ("Static Single Assignment"), traitant en plus de pointeurs multi-niveaux, et donnant plus d'information sur les pointeurs. Ensuite, une représentation compacte et effective des expressions symboliques est développée. Cette représentation préserve toutes les valeurs (valeurs intermédiaires incluses) d'une expression, d'où le fait que cette expression peut être comparée à une expression similaire de façon significative. Finalement, la possibilité de comparer des expressions symboliques (servant d'indices de tableaux) permet de comparer les réferences à des tableaux dans des blocs de base et des blocs hiérarchiques. Une analyse intitulée Analyse Hiérarchique de Dépendance des Tableaux génère de l'information, qui peut ensuite aider a la détection de instructions parallèles, et améliorer l'ordonnancement des instructions en réduisant le graphe de dépendance des données.

Le présent travail a été intégré à McCAT, le compilateur C optimisant et parallélisant, plus précisement a McADT, le testeur de dépendance des tableaux de

McCAT. A l'aide de ce cadre, plusieurs études expérimentales ont été réalisées dont les résultats sont présentés en détail.

Acknowledgments

This thesis would not be finished if it wasn't for the great help I got from my thesis advisor, Laurie J. Hendren. Not only was she helpful in technical matters, but also provided personal support and encouragement. Whether it was ideas about implementation, or help in publishing a paper, Laurie was there to listen and guide and in general just a great advisor. I would also like to thank the other professors of ACAPS (Advanced Compiler, Architecture and Parallel Systems) for providing great courses to build our understanding of computer architectures and compilers.

This work was financially supported by the Government of Canada via the Natural Sciences and Engineering Research Council (NSERC) as well as the Government of Québec via Fonds pour la formation de cherche et l'aide à la recherche du Quebec (FCAR).

Special thanks go to all the friends in ACAPS. Thank you for all the fun times as well as the help with McCAT. You will never be forgotten. Rakesh, thanks for introducing me to McCAT. You were the first. Ying, thanks for the discussion about loop parallelizations. You were one of the few I could brag about improvements to McADT that would be semi-interested. Shereen, thanks for proof reading this thesis. Ana, Chris, Justiani, Maryam, Luis, and Rakesh, thanks for making my summer NSERC scholarship at McGill a great time. It made it a much easier task to decide to join ACAPS after having met you. And thanks for developing McCAT so that working on this thesis was made possible. And to all the others just a general thanks.

And most importantly, I would like to thank my parents for emphasizing the importance of education. Thanks for your loving care, support and guidance during this time.

Dedicated to my parents

Contents

Abstract **ii**

Résumé **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Symbolic Array Dependence Analysis 2

1.2 Thesis Contributions 4

1.3 Thesis Organization 6

2 Background and Compiler Framework **7**

2.1 Structure of McCAT 7

2.1.1 SIMPLE intermediate representation 9

2.1.2 Data-flow analyses 10

2.2 McADT – McCAT Array Dependence Tester 15

3 Extended SSA Numbering: Defining a symbol **20**

3.1 Introduction 20

3.2 Static Single Assignment Form 22

3.3 Extended SSA Numbering 24

3.3.1 Difficulties with pointers 26

3.4 Definition and Implementation of Extended SSA Numbering 27

3.4.1 Implementation 29

3.4.2 Points of attention 30

3.5 Properties of Extended SSA Numbering 35

3.5.1 Factored use-def and def-use chains 35

3.5.2 Definition point properties 36

3.6 Defining "symbol" 39

4 Index Expression Analysis **40**
4.1 Motivating Example 41
4.2 Tree Representation 42
4.2.1 Possible Reaching Definitions 44
4.2.2 Working With Pointers 44
4.3 Induction Variables 45
4.3.1 Background on Induction Variables 46
4.3.2 Symbolic Variation of Induction Variable Analysis 48
4.4 Possible Symbolic Value Trees 51
4.4.1 Sum of Products 51
4.4.2 Preserving Intermediate Values 53
4.4.3 Compressing Tree Nodes 54
4.5 Building PSV tree algorithm 56

5 Symbolic Array Dependence Testing **59**
5.1 Possible Value Pairing 60
5.1.1 Building the plan 63
5.2 Symbolic Dependence Testing 65
5.2.1 Symbolic Simplifications 67
5.2.2 Symbolic ZIV and SIV array dependence tests 68
5.3 Satisfactory Answers 72

6 Hierarchical Array Dependence Analysis **77**
6.1 Introduction and Motivation 77
6.2 Analysis Objectives 78
6.2.1 Testable pairs 79
6.2.2 When to perform dependence tests? 79
6.3 Array Reference Summary 79
6.3.1 Assignment Statements 80
6.3.2 Statement blocks 80
6.3.3 Conditionals 80
6.3.4 Loops 81
6.3.5 Function Calls 82
6.4 Dependence Testing 82
6.4.1 No loop variables 83

6.4.2 Loop variables present . . . 83
6.5 Applications of Hierarchical Array Dependence Analysis . . . 84
6.5.1 Parallel Statement Detection . . . 84
6.5.2 Instruction Scheduling . . . 85
6.5.3 Scalar Replacement . . . 86
6.6 Conclusions . . . 87

7 Observations and Experimental Results **88**
7.1 Admissible Loop Nests . . . 89
7.1.1 Observations . . . 92
7.2 Induction Variables . . . 92
7.2.1 Observations . . . 94
7.3 Possible Symbolic Value Trees . . . 95
7.3.1 Observations . . . 99
7.4 Symbolic Simplifications . . . 99
7.4.1 Observations . . . 101
7.5 Symbolic ZIV and SIV Dependence Tests . . . 102
7.5.1 Observations . . . 103
7.6 Overall Results . . . 104
7.6.1 Observations . . . 106

8 Related Work **107**
8.1 Static Single Assignment Form . . . 107
8.2 Symbolic Analysis . . . 109
8.2.1 Symbolic Arithmetic . . . 109
8.2.2 Possible Symbolic Values . . . 109
8.2.3 Induction Variables . . . 110
8.3 Array Reference Summary . . . 113
8.3.1 Conservative Method . . . 113
8.3.2 Convex Hull Regions Method . . . 114
8.3.3 Atomic reference lists . . . 115
8.3.4 Linearized list . . . 115
8.3.5 Restricted Convex Regions . . . 115
8.3.6 Regular Sections with Bounds and Strides . . . 116
8.3.7 Hierarchical Array Dependence Analysis . . . 116

9 Conclusions and Future Work **117**
9.1 Conclusions 117
9.2 Future Work 119

A Other applications of Extended SSA Numbering **121**
A.1 Induction variables 121
A.2 Recognition of `ref` parameters 121
A.3 Strengthening reaching definitions 122
A.4 Reduction of Indirect References 123

B Decoding the McCAT ADT output **125**
B.1 Generated C dump 126
B.1.1 Extended SSA Numbering 126
B.1.2 Statement labels 126
B.2 Summary of array dependence information 126
B.2.1 Loop variable summary 126
B.2.2 Possible dependence info 127

List of Tables

2.1 Points-to relations for example program 12
2.2 Reaching definitions sets stored at each statement for example program. 13
2.3 Generalized constant propagation info for example program 13
2.4 Contents of read and write sets for example program 14
3.1 Description of benchmarks used to measure space requirements of DU, UD chains. 37
3.2 Average sizes of defs and uses sets stored at each variable reference. . 38
3.3 Example symbols that can be encountered in McCAT symbolic array dependence analysis framework. 39
7.1 Description of benchmarks used. 90
7.2 Effects of changes to admissible loopnest detection. 91
7.3 Summary of detected induction variables. 94
7.4 Sizes of PSV trees in admissible loops. 96
7.5 Sizes of PSV trees in hierarchical array dependence analysis. 97
7.6 Occurrence of array pairs in admissible loops that contained/used multi-node PSV trees. 98
7.7 Occurrence of array pairs in hierarchical array dependence analysis that contained/used multi-node PSV trees. 98
7.8 Effectiveness of symbolic subtract simplification. 100
7.9 Effectiveness of symbolic ZIV and SIV dependence tests. 103
7.10 Overall effectiveness of symbolic array dependence analysis. 105
A.1 List of benchmarks to which we applied the indirect reference reduction.124
A.2 Results of applying our indirect reference reduction transformation. . 124

List of Figures

1.1 Simple array dependence analysis example. 2
1.2 Example of array reference with symbolic index expression. 3
1.3 Example loop easily resolved independent using symbolic array dependence analysis. 4
2.1 Structure of the McCAT compiler. 8
2.2 Sample (a) C and (b) SIMPLE-C program for demonstrating data-flow analyses. 11
2.3 General structure of McADT 16
3.1 Example for comparing expressions with changing variable. 21
3.2 Example where comparing expressions with variables as symbols fails. 22
3.3 Example transformation of straight line code into SSA form. (a) original code, (b) code in SSA form. 23
3.4 Example transformation of code with `if` conditional into SSA form. (a) original code, (b) code in SSA form. 23
3.5 Program from Figure 3.2 in SSA form. 24
3.6 Example of SSA form vs. SSA Numbering. 25
3.7 Difficulties with pointer for SSA form shown in the original program, SSA form and SSA Numbered program. 26
3.8 Data accessible via a pointer. 28
3.9 Sample program Extended SSA Numbers. 29
3.10 Partial algorithm for SSA numbering analysis. 31
3.11 Effects of `break` statements in while loops. 32
3.12 Effects of `break` statements in do-while loops. 33
3.13 Example program showing SSA numbers for structures using both methods. 35
4.1 Index expression analysis in symbolic array dependence framework. . 40
4.2 Motivating example for index expression analysis. 41
4.3 First step of induction expression analysis: Build expression trees. . . 43

4.4 Expression tree for the expression $1 + x \times y$. 43
4.5 Expression tree with multiple possible definitions. 45
4.6 Second step of induction expression analysis: Induction variable analysis. 46
4.7 Example basic induction variable and its replacement. 46
4.8 Example induction variable patterns 48
4.9 Transforming IV pattern into expression in loop variables 50
4.10 Example of finished loop value for an induction variable. 51
4.11 Third step of induction expression analysis: Build possible symbolic value trees. 52
4.12 Representing Sum of Products. 53
4.13 Example comparing two expressions. 54
4.14 Example Possible Symbolic Value Trees. 55
4.15 Example "Compressed" Possible Symbolic Value Trees. 55
4.16 Algorithm for building PSV trees 57
5.1 Symbolic dependence testing in the framework of array dependence analysis. 60
5.2 Possible Value Pairing in the context of Symbolic Dependence testing. 61
5.3 Example loop with three array references with symbolic unknowns. . 62
5.4 Abstracting the structure of PSV trees for the purpose of building dependence test plan. 63
5.5 Algorithm to generate Possible Value Pairings 64
5.6 Symbolic Simplifications and ZIV and SIV tests in the context of Symbolic Dependence testing. 66
5.7 Example of symbolic subtraction simplification for the symbolic value pair "`i+2*j+x+y`" vs. "`2*i-j+x+y+1`" 68
5.8 Example output from McCAT analyzing a symbolic ZIV dependence. 70
5.9 Example output from McCAT analyzing a Strong-SIV dependence with symbolic distance. 71
5.10 Example output from McCAT analyzing a Weak-Zero-SIV dependence with symbolic distance. 72
5.11 Example output from McCAT analyzing a Weak-Zero-SIV dependence with symbolic distance. 73
5.12 Example output from McCAT analyzing a Weak-SIV dependence with symbolic distance. 74
5.13 Module for deciding on a satisfactory answer in the context of symbolic dependence testing. 75

6.1 Example loop with initialization code. 77
6.2 Example code where all statements should be determined independent. 78
6.3 Example admissible loop nest to illustrate array reference summary for loops. 81
6.4 Admissible loop constructed to test "`A[i+1]` where `i` in [0,n]" vs. "`A[0]`". 83
6.5 Checking Bernstein condition for two assignment statements. 84
6.6 Example of using hierarchical array dependence information to better schedule two statements interfering at source statement level. 85
6.7 Two versions of scalar replacement optimizing transformation. 86
7.1 Situation where "good" loop detection needs to continue towards inner loops. 90
7.2 Example loop from `clinpack` benchmark. 92
7.3 Example loop that requires PSV trees to perform most accurate array dependence analysis. 95
7.4 Example loops where symbolic subtraction is common. 102
7.5 Example of strip-mining when dependence information gives symbolic dependence distance. (a) original loop, (b) transformed loop. 104
8.1 Example symbolic expression with possible values. 109
8.2 Example program from Figure 8.1 in SSA form. 110
8.3 Difference table for f. 112
8.4 Extended difference table for f. 112
8.5 Example of three array references. 114
8.6 Conservative whole array summary. 114
8.7 Convex hull region summary.. 114
8.8 Atomic list of array location referenced. 115
8.9 Linearized list of array locations referenced. 115
8.10 Restricted Convex Regions summary 116
8.11 Regular Sections with Bounds and Strides summary. 116
A.1 Example of reaching definitions for pointer references. 122
A.2 Two transformations to reduce indirect references. 123
B.1 Example output from McCAT symbolic array dependence analysis. . 125
B.2 Example PSV as output by McCAT and its graphical representation. 128

Chapter 1

Introduction

Array dependence analysis is a key component of any parallelizing/vectorizing compiler. A well known 90-10 rule supports the importance of array dependence analyzers.

> *A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code.*[HP90]

The only way this rule can be true is that programs must spend significant time in some type of iteration or repetitive execution. In imperative languages such as C and FORTRAN, structured iterative constructs are provided, as well as convenient built-in data structures that work well in iterative situations. In FORTRAN these constructs are `DO` loops and arrays, whereas in C they are `for` loops and arrays.

Array dependence analysis is an analysis that tries to determine dependencies due to array references inside loops. Array dependence analysis is so important because it provides compilers with information for parallelization and strategic code generation in the areas where most benefit can be gained (where 90% of time is spent). Providing information about loop dependencies, allows the compiler to better schedule loop execution or perhaps parallelize/vectorize loops if such specialized hardware is available.

Dependence analysis has been the topic of a lot of research [AK87, Ban88, Fea91, GKT91, Hag90, MHL91, Jus95]. A typical use of dependence analysis is illustrated by the loop in Figure 1.1. In this example, dependence analysis is used to determine that there is a flow dependence from statement *S1* to statement *S2* with dependence distance of (4). This means that the array reference at statement *S2* will use the data that was generated 4 iterations before at statement *S1*. (Please refer to [Wol96] for more detail on general array dependence analysis.)

```
        for(i=0;i<100;i++)
        {
S1:        A[i+4] = ...;
S2:        ... = A[i];
        }
```

Figure 1.1: Simple array dependence analysis example.

Since the development of the initial dependence analyses, many refinements have been developed that allow array dependence testers to handle more and more complex situations. Recognition of induction variables [Wol92, GSW95] permitted many more situations to be analyzed. Closer to home, Justiani [Jus95, JH94] implemented array dependence analysis for a C optimizing/parallelizing compiler McCAT. This required overcoming many of the difficulties of C's pointers.

1.1 Symbolic Array Dependence Analysis

An area of array dependence analysis that has yet to be fully developed is the situation when array index expressions contain a variable of unknown value (symbols). Normally, in order for array dependence analysis to compute the dependencies due to arrays, it must have, for each array reference, index expressions that contain only loop variables and integer constants. For example, consider the array reference `A[i+3][temp]` in an loop nest with loop variables `i` and `j`. For the first dimension, the index expression contains one loop variable, namely `i`, and an integer constant `3`. This index expression is fine. Consider the second dimension. The variable `temp` cannot be accepted as an analyzable index expression. To try and resolve such an index expression, an array dependence tester must follow reaching definitions to try and come up with a replacement expression [JH94]. This works in most cases, and it has satisfied most compilers, but there can still be situations where the resulting index expression is still not satisfactory. Consider the example in Figure 1.2.

In this example the array reference in statement *S5* is of interest. The second dimension has index expression `temp`. The symbol `temp` is not acceptable for array dependence testing, but this is easily resolved by searching for the reaching definitions of `temp` (statement *S4*). This preparation step can replace `temp` by `j + x`. Variable `j` is a loop variable but `x` is neither a loop variable or an integer constant. Many array dependence analyses have turned to other analyses such as constant propagation [VCH96, WZ91] in hopes that the variable `x` is resolved into a constant. This may happen considering the structure of statement *S3*, but assuming that the value of `x` is

```
S3:  x = ...;
     for(i=0;i<100;i++)
       for(j=0;j<100;j++)
       {
S4:      temp = j + x;
S5:      A[i+3][temp] = ...;
S6:      ... A[i+3][j] ...;
       }
```

Figure 1.2: Example of array reference with symbolic index expression.

unknown at compile time, the array dependence tester will not be able to determine dependencies for this array reference. Further tricks can be applied, but there will always be situations where the index expression contains a variable whose value is unknown leading to a forced conservative assumption of a possible dependence.

Symbolic array dependence analysis is able to determine dependencies due to arrays even in the presence of variables with unknown value. For example, symbolic array dependence analysis can state that there exists a dependence between statements *S5* and *S6* with dependence distance (0,-`x`). Note that the information is not as precise because it is not always possible to determine if the dependence is a flow-dependence or anti-dependence, because we don't know if `x` is positive or negative. Also, the dependence distance is symbolic and therefore code generation based on this information is slightly more difficult.

Some situations requiring symbolic framework for array dependence analysis do give as exact and useful information as "regular" array dependence analyses. Take for example a loop nest extracted from a benchmark in Figure 1.3. This loop illustrates a typical situation which is easily resolved by symbolic array dependence analysis. The situation arises due to an unanalyzable outer loop[1], in this case a `do-while` loop. This outer loop maintains an induction variable, in this example `j`. The inner loop is accepted by the array dependence analyzer and detects several array reference pairs. Each array reference is two-dimensional. One of the dimensions is exclusively referenced using the loop variable of the outer loop. In this example, this results in dependence tests such as "`i`" vs. "`i`" where `i` is a constant symbol (for the duration of the inner loop) of unknown value. The constant symbol `i` is not a problem. Consider a test pair "`5`" vs. "`5`". The dependence distance would be (0). Similarly, the dependence distance would be (0) for "`6`" vs. `6`", "`7`" vs. `7`", or any integer pair "k_1" vs. "k_2'" as long as $k_1 = k_2$. Now, let k_1 be `i` from one array reference and k_2 be `i`

[1]Some compilers detect such loops as structured loops and are capable of analyzing them.

from the second, when performing the dependence test "`i`" vs. "`i`". The condition that $k_1 = k_2$ is satisfied and therefore we can conclude a dependence distance of (0). This particular symbolic simplification, turns out to be very useful and can be frequently applied.

```
i = 1;
do
{
  for (j=2; j<n; j++)
  {
    aa[i][j] = aa[i][j-1]*bb[i][j-1];
    bb[i][j] = aa[i][j-1]*bb[i][j-2];
    cc[i][j] = cc[i][j-1]*cc[i][j-2];
  }
  i++;
} while (i < n)
```

Figure 1.3: Example loop easily resolved independent using symbolic array dependence analysis.

1.2 Thesis Contributions

The final goal and major contribution of this thesis it the development of a symbolic framework for array dependence analysis. This framework consists of

- definition of a symbol;
- effective representation of a symbolic expression; and,
- effective comparison of two symbolic expressions.

The complete framework works for the full C programming language, and has been implemented in the McCAT C compiler. The result of this analysis are loops with a summary of dependencies due to array references including dependence distance vectors possibly containing symbolic expressions.

On the way to the final goal of symbolic array dependence analysis, many subproblems had to be resolved. Many of the contributions of this thesis are the resolutions of these problems, and some tangent topics that surfaced due to these solutions. These other contributions are itemized below.

- Providing properties of Static Single Assignment (SSA) form to languages with multi-level pointers (*Extended SSA Numbering*). In developing symbolic dependence testers it is necessary to determine when two references to the same variable denote the same value. Thus, the property of "same name, same value" in SSA form is very useful. Unfortunately, in languages with pointers, the simple SSA form is not adequate. This thesis develops a new numbering scheme, called *Extended SSA Numbering*, that works in the presence of pointers.

- Out of developing extended SSA numbering, several uses of the SSA properties for pointers were exploited to help out in other analyses and transformations. These were: (1) a transformation similar to scalar replacement to reduce indirect references, and (2) an improvement for reaching definitions dataflow analysis in situations using indirect references.

- Standard induction variable analysis expects the induction variable to have a clearly defined initial value, as well as an increment whose values need to be known at compile time. With little difficulty, symbols can be used in the place of these constants. For example, with symbolic capabilities, induction variables that start with an unknown symbolic value can be determined as well as induction variables that are incremented by a constant, known only at run time, can be analyzed. Prior work on this topic has been done; however, without an array dependence analysis that can work with symbols, such effort offers little optimizing opportunity.

- Expressions in a program can be determined, by the compiler, to have several possible values. There may be many ways to express a symbolic expression just due to equivalences introduced by assignment statements. Thus, effectively representing a symbolic expression is a challenge. In this thesis, I present a way to represent a symbolic expression accurately and compactly.

- In addition to representing symbolic expressions effectively, one must derive a plan to to compare symbolic expressions in a way that does not miss any important facts, as well as not consuming exponential computing power. The internal representation is compact enough so that it increases comparison time by at most 4 times, independent of expression complexity.

- Having developed the ability to perform dependence tests for array references with symbolic unknowns, it becomes possible to perform array dependence tests of array references within basic blocks. This can help out in instruction scheduling as well as improve optimizations such as scalar replacement. The symbolic framework was used to summarize array references in each statement. This information was then used to determine dependencies due to array references for statements in any basic block or same hierarchical block. Knowing these dependencies allows for better code generation.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 describes the background and starting point for my work. It describes the structure of the McCAT compiler for which this symbolic framework was written and its analyses that were needed to implement the symbolic analysis. It also describes the McCAT array dependence tester, its components and modules which were modified to implement symbolic array dependence analysis. Chapter 3 describes the extended SSA numbering analysis and how the symbolic framework defines "symbol". Chapter 4 investigates how to represent array index expressions. It discusses the basic concept of following reaching definitions, followed by refining the expression using induction variable analysis, and then representing the symbolic value effectively and compactly for the purposes of comparison. Chapter 5 describes array dependence tests for symbolic index expressions. It presents a symbolic simplification that tries to reduce symbolic test pairs into expressions without symbols, as well as presents a symbolic zero-index-variable dependence test, and a set of symbolic single-index-variable dependence tests. Chapter 7 shows how the components of the symbolic framework for array dependence analysis perform for real applications and synthetic benchmarks. This chapter mostly concentrates on showing the importance of adding symbolic capabilities to an array dependence tester instead of arguing the importance of an array dependence tester in a parallelizing/optimizing compiler. Chapter 8 introduces some related work done by others in the field of array dependence analysis especially as it relates to the contributions of this thesis. Chapter 9 presents the conclusions of this thesis and any future work that may be done on improving the symbolic array dependence analysis.

In addition to the main chapters two appendices are provided: Appendix A discusses additional uses of extended SSA numbering presented in Chapter 3, and Appendix B gives a brief guide on reading the output generated by the symbolic array dependence analysis.

Chapter 2

Background and Compiler Framework

In this chapter, I present the structure and components of the McCAT optimizing/parallelizing compiler on which my work with symbolic array dependence testing is based. Also I present an outline of the symbolic analysis framework and how it fits in the McCAT compiler.

Readers familiar with McCAT can skip section 2.1, but are encouraged to read section 2.2 to see how the work of this thesis fits into McCAT.

2.1 Structure of McCAT

The McCAT optimizing/parallelizing compiler is a part of an on going project called McCAT (McGill Compiler/Architecture Testbed). The compiler is equipped with many analyses and transformations as well as a simple intermediate representation called SIMPLE [HDE+92, Sri92].

The structure of the McCAT compiler is illustrated in Figure 2.1. McCAT accepts standard C programs in one or more files. For the purposes of inter-procedural analyses, all the input C files are first parsed and symbolically linked into one. Before performing any analyses or transformations, the standard C program is simplified to a simplified C called SIMPLE-C. High-level transformations and analyses are then performed at the SIMPLE level, shown in Figure 2.1 as a box listing several of the analyses. All of the components of the Symbolic Array Dependence Testing framework reside at this level of the compiler. For example, extended SSA numbering and array index expression analysis are two components of the symbolic analysis framework that appear in Figure 2.1.

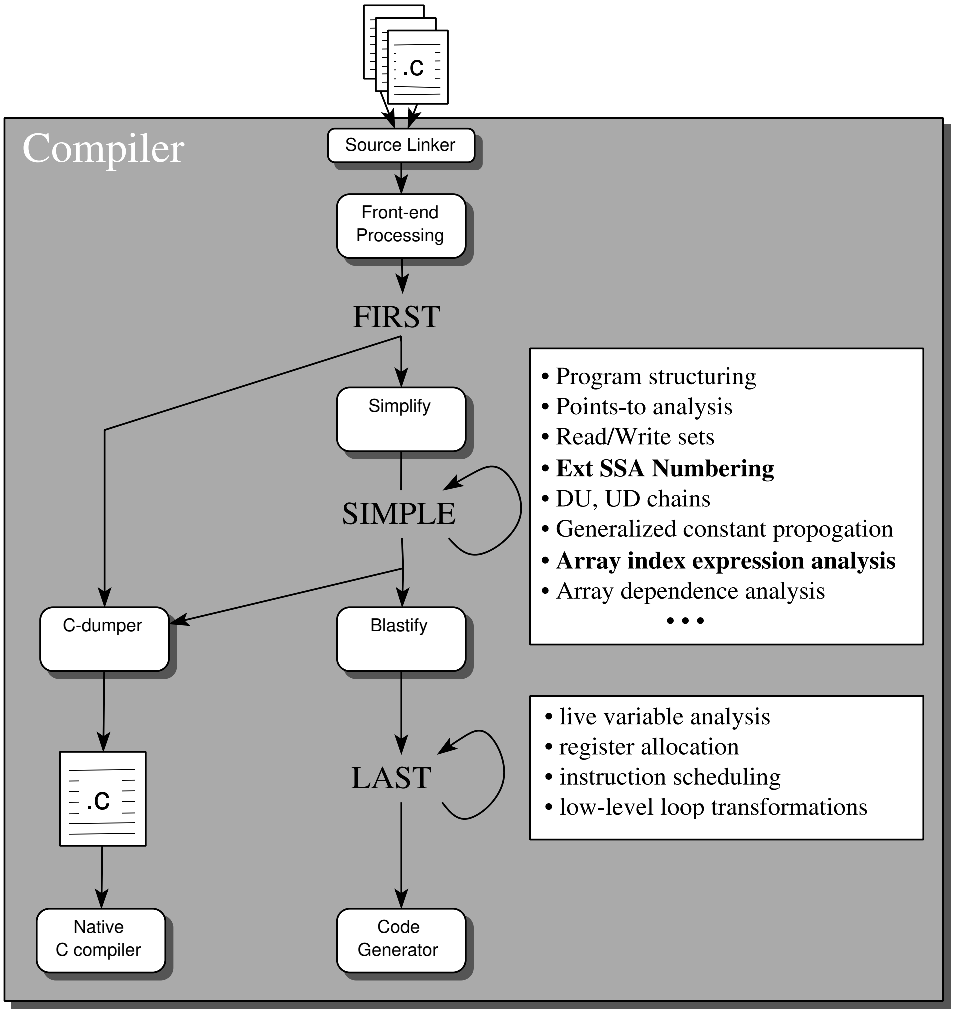


Figure 2.1: Structure of the McCAT compiler.

At any stage of performing analyses and transformations, at the SIMPLE level, the compiler can output the simplified C program with analysis information present as comments in the C program. Typically this is done only after all SIMPLE level analyses have finished and all SIMPLE transformations have optimized the program. In Figure 2.1 this stage is called the C-dumper.

In addition to analyses and transformations at the high-level intermediate representation, McCAT creates a low-level intermediate representation called LAST

[HDE+92, Don94]. This low-level intermediate representation ensures that it has links to SIMPLE in order to take advantage of the information of the higher level. At the LAST level, analyses and transformations such as live variable analysis, register allocation, common sub-expression elimination, and instruction scheduling are performed before code is generated for a Load/Store architecture.

The remainder of this section discusses, in further detail, the SIMPLE intermediate representation, and analyses/transformations needed by the symbolic array dependence testing framework.

2.1.1 SIMPLE intermediate representation

What makes McCAT most attractive, is its simple intermediate representation. The C language can be very complex to analyze, due to the high degree of freedom in writing expressions and control structures. One of the first steps of McCAT is to transform C input programs into SIMPLE-C. SIMPLE-C is expressively equivalent to C with the exception that complex structures are replaced by a series of simpler statements and structures.

The following is a list of key simplifications that are performed by McCAT.

- Complex expressions are simplified to a series of statements containing at most one operator.

  ```
  x = a op b;
  x = op a;
  ```

- All multi-level indirections are simplified to a series of single-level indirections.

  ```
  x = **p;   ==>   temp1 = *p;
                   x = *temp1;
  ```

- All array indexes and function arguments are simplified to be a single identifier reference.

  ```
                         temp1 = i + 1;
  x = A[i+1][j+c];  ==>  temp2 = j + c;
                         x = A[temp1][temp2];
  ```

- Complex loop initializers, conditions and iterators are reduced to simple single statements or expressions with no side effects, according to a chosen standard, by introducing statements before the loop and at the end of the loop body.

```
                                        j = 0;
for(j=0,i=0; i<10; j++,i++)             for(i = 0; (i <= 9); i = (i + 1))
{                              ==>      {
  /* Body */                              /* Body */
                                          j = (j + 1);
}                                       }
```

- *Program structuring* appearing as the first transformation at the SIMPLE level in Figure 2.1 removes all unstructured control flow branches (`goto`'s) [EH94, Ero95]. This is an important phase that makes all analysis of programs much simpler.

An example of simplifying a program is available in Figure 2.2. Part (a) shows the input program and part (b) the SIMPLE-C version of the program. Additional information about SIMPLE-C such as the language grammar is available in [Sri92].

2.1.2 Data-flow analyses

In addition to program structuring and simplification McCAT has several data-flow analyses. Below I list four of these data-flow analyses that are important in setting up the symbolic framework for array dependence testing. These are points-to analysis, reaching definitions, generalized constant propagation, and read/write sets. To illustrate these analyses at work, the SIMPLE-C program in Figure 2.2 is used. Figure 2.2(a) shows the input C program segment and Figure 2.2(b) shows the SIMPLE-C version of this same program.

Points-To analysis

Points-to analysis[EGH94] identifies for each point in the program, the relationships between pointers and named locations. The relationship is represented by a pair (`p`,`v`) of a pointer `p` and a variable `v` indicating that `p` points to `v`. In the case where a pointer `p` at some program statement can point to `v` but may also point to some other location, the relationship is flagged as may-point-to `v`. This is represented by a pair with a question mark following it, (`p`,`v`)?

Table 2.1 shows the points-to relationships at program points just before labeled statements of the SIMPLE-C program in Figure 2.2(b). For example, statement *S8* introduces a new points-to relation, namely that pointer `q` points to the variable `a`. This relation is not changed in the program and therefore appears in all statements thereafter. Also notice that the points-to relation for the pointer `p`, at statement

```
int f(int a)
{
  int *p,*q,**r;
  int b,c,d;


  q = &a;
  c = 10;
  if(a < 10)
  {
    b = 1;
    c = a;
    p = &b;
    r = &p;
  }
  else
  {
    b = 5;
    p = &c;
    r = &q;
  }


  d = **r * 5;
  return d;
}
```

(a)

```
S7:  int f(int a)
     {
       int *p, *q, **r;
       int b,c,d;
       int temp_0,*temp_1;

S8:    q = (& a);
S9:    c = 10;
S10:   if ((a <= 9))
       {
S11:     b = 1;
S12:     c = a;
S13:     p = (& b);
S14:     r = (& p);
       }
       else
       {
S15:     b = 5;
S16:     p = (& c);
S17:     r = (& q);
       }
S18:   temp_1 = (* r);
S19:   temp_0 = (* temp_1);
S20:   d = (temp_0 * 5);
S21:   return (d);
     }
```

(b)

Figure 2.2: Sample (a) C and (b) SIMPLE-C program for demonstrating data-flow analyses.

S18, is a may relation. This is because on different paths through the conditional, the pointer `p` was assigned the address of different variables. After the conditional, starting with statement *S18*, the pointer `p` points to either `b` or `c`.

Points-to analysis is crucial in many steps of array dependence analysis in order to handle situations where pointers are involved.

Reaching Definitions

Reaching Definitions[ASU86] is an analysis that tells us which definition of a variable `v` reaches the particular use of `v`. Depending on the intermediate representation, information of this analysis is stored in different ways. In McCAT, at each statement,

Stmt	Points-to relations
S8	
S9	(q,a)
S10	(q,a)
S11	(q,a)
S12	(q,a)
S13	(q,a)
S14	(p,b) (q,a)
S15	(q,a)
S16	(q,a)
S17	(p,c) (q,a)
S18	(r,p)? (r,q)? (p,c)? (p,b)? (q,a)
S19	(temp_1,a)? (temp_1,b)? (temp_1,c)? (r,p)? (r,q)? (p,c)? (p,b)? (q,a)
S20	(temp_1,a)? (temp_1,b)? (temp_1,c)? (r,p)? (r,q)? (p,c)? (p,b)? (q,a)
S21	(temp_1,a)? (temp_1,b)? (temp_1,c)? (r,p)? (r,q)? (p,c)? (p,b)? (q,a)

Table 2.1: Points-to relations for example program

a set is attached. This set contains all the definitions that reach the statement. To reduce space consumption, only definitions of variables appearing in the statement are stored. (i.e. One will not ask for reaching definitions of variable `v` if the statement has nothing to do with `v`.)

For example, consider the short program in Figure 2.2(b). Statements *S11* and *S15* both define the variable `b`. If one asks for the reaching definitions of `b` at statement *S18*, the answer will be the set { *S11*, *S15* }. Table 2.2 shows the reaching definitions sets stored at each statement. Note the reaching definitions sets for the variable `a`. Because the value of `a` comes from the mapping of actual arguments to formal parameters, its definition is the function header. Also, because in McCAT reaching definitions is intra-procedural, global variables will have initial values coming from the function header indicating that the definition is outside.

Reaching definitions are mostly used by the symbolic framework to build full expressions for array index values.

Generalized Constant Propagation

McCAT is equipped with inter-procedural constant propagation [VCH96, WZ91]. Not only does this analysis propagate constants, but it also propagates ranges of values instead of just constants. The analysis can associate with a variable occurrence a single constant value (eg. 5), a range of values (eg. [0,5]), or an open range (eg.

Statement	Reaching Definitions Sets
S8	a{*S7*}
S9	
S10	c{*S9*}
S11	
S12	a{*S7*}
S13	b{*S11*}
S14	p{*S13*}
S15	
S16	c{*S9*}
S17	q{*S16*}
S18	r{*S14*, *S17*} p{*S13*, *S16*} q{*S8*} a{*S7*} b{*S11*, *S15*} c{*S9*, *S12*}
S19	temp_1{*S18*} a{*S7*} b{*S11*, *S15*} c{*S9*, *S12*}
S20	temp_0{*S19*}
S21	d{*S20*}

Table 2.2: Reaching definitions sets stored at each statement for example program.

$(-\infty$,5], [5,∞)).

Table 2.3 shows the constant propagation information for the program in Figure 2.2(b). Consider two examples. First, look at line *S11* and *S15*. Variable `b` is set to

Statement	Generalized Constant Propagation Info		
S8			
S9			
S10	c $\in [10,10]$		
S11	c $\in [10,10]$	a $\in (-\infty,9]$	
S12	c $\in [10,10]$	a $\in (-\infty,9]$	b $\in [1,1]$
S13	c $\in (-\infty,10]$	a $\in (-\infty,9]$	b $\in [1,1]$
S14	c $\in (-\infty,10]$	a $\in (-\infty,9]$	b $\in [1,1]$
S15	c $\in [10,10]$	a $\in [10,\infty)$	
S16	c $\in [10,10]$	a $\in [10,\infty)$	b $\in [5,5]$
S17	c $\in [10,10]$	a $\in [10,\infty)$	b $\in [5,5]$
S18	c $\in (-\infty,10]$	a $\in [10,\infty)$	b $\in [5,5]$
S19	c $\in (-\infty,10]$	b $\in [1,5]$	
S20	c $\in (-\infty,10]$	b $\in [1,5]$	
S21	c $\in (-\infty,10]$	b $\in [1,5]$	

Table 2.3: Generalized constant propagation info for example program

constant integer values. We see the effect of these statements by the appearance of `b` $\in [1,1]$ and `b` $\in [5,5]$ at lines *S12* and *S16*. These two assignments occurred in two separate data flow paths. When these two paths merge at statement *S18* the merge of these two constants happens resulting in `b` $\in [1,5]$. If ranges were not considered then `b` would be unknown at statements *S18*, *S19*, *S20*, *S21*.

Second, consider the variable `a`. Its value is unknown, but inside the paths of the conditional we know that `a` $\in (-\infty, 9]$ or `a` $\in [10, \infty)$. We know this because of the condition expression in the `if` statement at line *S10*. This second example demonstrates how McCAT generates open ranges for some variables.

Generalized constant propagation is used extensively to help to reduce the number of symbolic unknowns as well as to provide some range information for symbolic unknowns that remain.

Read/Write Sets

Read/Write sets (also, known as Mod/Ref sets) is a very useful inter-procedural analysis that states which named locations are read/written by the statement. For example, for a simple assignment statement, the variable on the left hand side is written and the variables of the expression on the right hand side are read. The analysis extends this idea hierarchically to block statements, conditionals, loops and function calls.

Table 2.4 gives the contents of the read and write sets of each statement from the example program in Figure 2.2(b). Notice that the read/write sets include may

Statement	Read Set	Write Set
S7	a	
S8		q
S9		c
S10	a	b, c, p, r
S11		b
S12	a	c
S13		p
S14		r
S15		b
S16		p
S17		r
S18	r, q, p	temp_1
S19	temp_1, a, b, c	temp_0
S20	temp_0	d
S21	d	

Table 2.4: Contents of read and write sets for example program

read and may write information. This is seen in statement *S18*. When analyzing the reference `*r`, it is not known if `r` will be pointing to `p` or `q` so the read set contains both of them. A similar situation happens at statement *S19* when referencing `*temp_1`. Points-to information states that `temp_1` points to either `a`, `b`, or `c`; hence, all are put in the read set. Also, notice that while dereferencing a pointer the value held by the pointer (an address) is also being read. This means that the pointer should also appear in the read set.

This analysis is most useful in determining which variables are written by a block of code. This information is available because read/write sets analysis is hierarchical in depositing its information. For example, one may be interested in what variables are written by the entire `if` construct. This information is easily available by just looking at the write set attached to the `if` statement (*S10*). The information retrieved is that variables `b` ,`c`, `p` and `r` are written.

2.2 McADT – McCAT Array Dependence Tester

Having described in general the structure of McCAT compiler it is necessary now to look more closely at array dependence analysis as implemented in McCAT. In addition to examining what framework exists to support array dependence analysis, the redesigned and re-implemented components to support symbolic analysis are described.

The array dependence analysis framework for McCAT (called McADT for McCAT Array Dependence Tester) was written by Justiani. More detailed information about the structure of McADT can be found in [JH94, Jus95]. The general structure of McADT is shown in Figure 2.3.

Each of the components of McADT, shown in Figure 2.3, is a step decorating the Abstract Syntax Tree (AST) representing the input program with additional information. First, a classification of loops of the input program selects loops with "good" behavior and decorates their AST node with information such as lower and upper bounds as well as the name of the loop variable. Next, a pass through each loop body is made to collect all the statements containing array references and attaches this list to the decorated "good" loop AST node. A pass through each of the statements with array references can be then done applying *index expression analysis* decorating each array reference with an internal representation of the index expression. Following that, array references appearing in each "good" loop are paired for possible dependences. And finally, each array pair is visited with all index information available and

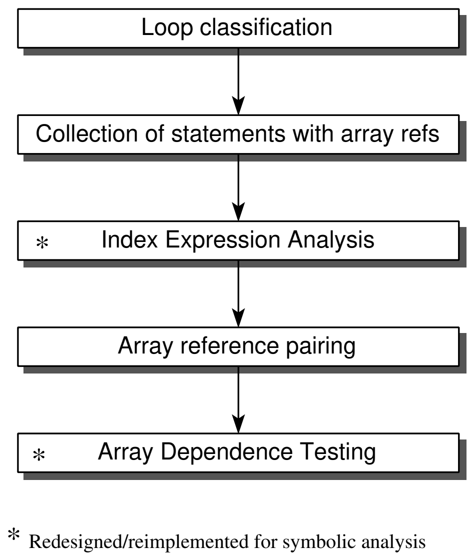


Figure 2.3: General structure of McADT

tested using one or several dependence tests [Wol96, GKT91, MHL91].

Having described briefly the process as a whole let's take some time to examine each individual step.

Loop Classification

The first step of McADT is loop classification. With large freedom given to the user of a language such as C, there are many ways one can write loops that can be very hard or even impossible to analyze for array dependences. For this reason McADT scans all loop constructs of the input program to determine if the loop behavior is understandable enough for dependence tests to be applied. The conditions are as follows [Jus95]:

- There is only one loop variable for every for-loop header.
- Lower/upper bounds of loop variables can only be either constants or variables not modified by the loop body.
- The innermost body of loop-nests consists of only assignment statements and/or conditional statements such as if-then-else statements or switch-case statements.

Loop constructs, `goto`, `break`, and `continue` statements are not allowed. Function calls, in general, are not allowed.

- Any assignment statements between two consecutive for-loop headers or between two consecutive for-loop exits are allowed as long as they do not contain array references.

- There is no modification or reassignment of loop variables and their aliases by any of the statements in the loop-nests. Initialization and incrementing of the loop variables must be the only statements in the for-loop header.

In addition to identifying "good" loop constructs, loops are normalized [Jus95]. This means that the loop is transformed so that the loop variable starts with the value 0 and is incremented by 1 at the end of each iteration. To expand the number of good loops accepted, I have expanded the loop normalization module so that it can preserve points-to and read/write sets dataflow information as it inserts new statements. With this accomplished, "good" loop detection, which is closely coupled with loop normalization, can be performed after points-to and read/write sets are computed. With read/write information at the time of detecting "good" loops, it is possible to determine if a function call can be accepted in a "good" loop. This expands the number of loop nests that can be tested.

At the end of this stage all loops identified as candidates for analysis have the following properties.

- The loop variable starting value is 0.

- The loop variable is incremented exactly by 1 and only once at the end of the loop body.

- The body of the loop does not contain function calls that could affect values of loop variables or arrays.

- The loop body can only contain assignment statements, conditionals or other "good" loops. Loop structures such as `while` and `do-while` will not appear in the loop body.

- In a loop nest, array references are allowed only in the innermost loop body.

Array Statement Selection

This stage is very simple but it makes all following stages simpler. In this stage the body of each selected loop is scanned to select statements containing array references. Array references are collected and enumerated. This list of statements and array references is made easily available at the loop header, relieving other components the trouble of scanning the loop body.

Index Expression Analysis

The purpose of this component is to analyze the index expression of each array reference. The result of the analysis is an internal representation of the index expression that facilitates performing array dependence tests. Typical steps taken by this component are building a complete expression tree of the index expression, performing induction variable analysis [Wol92, GSW95], and finally representing the expression in canonical form.

Expressions in canonical form must be written as $e_1 i_1 + e_2 i_2 + \ldots + e_n i_n + e_{n+1}$ where i_j are loop variables, and e_j are expression not involving any loop variables. Clearly we can construct expressions that can't be put into canonical form (eg. $i_1 i_2$). For this reason some array references are rejected at this stage. For these references a conservative assumption is made and any dependences involving this reference are automatically declared to possibly exist.

This component of McADT had to undergo complete redesign and new implementation to facilitate effective testing in the presence of symbolic expressions. A major part of this thesis is the redesign and implementation of this component. A complete discussion of this part is given in Chapter 4.

Array Reference Pairing

This component pairs together array references that can possibly generate dependences. In general, one can say that a dependence exists among two array references at statements S and T if the following conditions apply[Wol96]:

1. At least one of the two array references is on the left hand side of the assignment statement.

2. The two array references in question refer to the same array or are possibly aliased to the same array.

3. There exist two iteration vectors $\vec{\imath}$, $\vec{\jmath}$ such that $S_{\vec{\imath}}$ and $T_{\vec{\jmath}}$ reference the same element of the array.

4. There does not exists an iteration vector $\vec{k}$ between the iterations $\vec{\imath}$ and $\vec{\jmath}$ in sequential execution, that overwrites the value stored at the common array element determined in step 3.

Most compilers don't check the last condition and conservatively assume it to be true. McCAT with McADT does the same. This stage of McADT collects only those array reference pairs for which conditions 1 and 2 apply. The next component checks condition 3.

Array Dependence Testing

This last component of McADT is the heart of any Array Dependence Tester. This is the component that examines the index expressions in detail, testing if there is any possibility that the same elements of the array are referenced. Possible dependences are classified and tests such as ZIV, SIV, MIV and Delta test [GKT91] are used to prove or disprove dependence. The results of these dependence tests are then deposited with the array pairs and attached to the AST for other analyses and transformations to use.

This component of McADT also had to be modified so that it can work with symbolic data. Symbolic versions of some dependence tests were written and added to the framework of tests made available by McADT. The implementation of the symbolic versions of dependence tests, as well as developing strategies to best test array reference pairs are described in Chapter 5.

Chapter 3

Extended SSA Numbering: Defining a symbol

To define symbolic analyses it is necessary to first define what a *symbol* is in the context of a C program. In this chapter, I establish this definition of a symbol. Further development of the symbolic dependence analysis framework will only deal with symbols instead program variables.

In addition, in the process of establishing a definition for a symbol, a very interesting and useful analysis, called *Extended SSA Numbering*, was developed. This analysis can be very useful to an optimizing compiler and some of its uses are discussed in appendix A.

3.1 Introduction

One of the goals of the symbolic framework is to be able to compare expressions in the presence of variables with unknown value. In mathematics, we often write expressions and equations using symbols. For example:

Expression 1: $x + y$
Expression 2: $x + y + 1$

In the context of mathematics, if one is asked to compare the two example expressions, one would proceed by establishing the possible equality.

$$x + y \stackrel{?}{=} x + y + 1$$

The next step would be to cancel the occurrence of x from both sides of the equation followed by y.

$$\begin{aligned} y &\stackrel{?}{=} y+1 \\ 0 &\stackrel{?}{=} 1 \end{aligned}$$

Because 0 is not equal to 1, it can be concluded that the expressions were not equal.

Can this method of comparison be extended to the context of computer programs? Consider the simple example program with two expressions in the context of array references in Figure 3.1.

```
     for(i=0;i<10;i++)
     {
       x = x + 1;
       ...
       expr1 = x + y;
       expr2 = x + y + 1
S22:   A[expr1] = A[expr2];
       ...
     }
```

Figure 3.1: Example for comparing expressions with changing variable.

At first glance it appears that it is possible. If the variable names were used for mathematical symbols the same process would be used and the same conclusion drawn. Start comparing `expr1` with `expr2` from statement *S22*. Then expand their meaning and start cancelling symbols.

```
expr1  ?=  expr2
x + y  ?=  x + y + 1
    y  ?=  y + 1
    0  ?=  1
```

This may be true for the basic block, the loop body, but not true across iterations of the loop, because the value of `x` changes from iteration to iteration. Consider even a situation outside of a loop context, Figure 3.2.

In this example statement *S26* referencing two elements of array `A` requires the comparison. Again starting with `expr1` and `expr2`, expand their meaning from their definitions in statements *S23* and *S25*. The result is the same two expressions `x + y` and `x + y + 1`. Applying the process of comparison leads to a conclusion that the

```
     ...
S23: expr1 = x + y;
     ...
S24: x = ...;
     ...
S25: expr2 = x + y + 1;
     ...
S26: A[expr1] = A[expr2];
```

Figure 3.2: Example where comparing expressions with variables as symbols fails.

two expressions differ by 1. This is not necessarily true because of statement *S24* that redefines the value of `x` to possibly something different.

To conclude, using variable names as symbols will not preserve mathematical simplification rules for relations. It is necessary to come up with a different definition of "symbol" in the context of computer programs. Using Static Single Assignment (SSA) form one can define symbol in a way that will solve the problem in the context of basic blocks and conditional statements such as `if` and `switch`. The context of loops cannot be solved, but in some cases induction variable analysis, described in section 4.3, can help.

The remainder of this chapter is organized as follows. Section 3.2 briefly describes static single assignment form and how it could be used to define a symbol. Section 3.3 discusses the problems one encounters trying to apply SSA form to languages with multi-level pointers and how I propose to solve them using extended SSA numbering. Section 3.4 describes how to implement extended SSA numbering, and section 3.5 describes the properties of extended SSA numbering. Finally, in section 3.6, I define the meaning of "symbol" for the remainder of this thesis.

3.2 Static Single Assignment Form

Static single assignment (SSA) form is an intermediate representation widely used in FORTRAN compilers. It is an intermediate form that facilitates easy design and implementation of analyses and transformations. SSA form and its properties have been extensively studied [AWZ88, BM94, CFR+89, CFR+91, RWZ88, SG95].

The idea of SSA form is very simple:

- Every new definition of a variable will get a new name.

- In situations where two definitions meet, a merge is explicitly specified using a ϕ-node and the result is put into a new variable.

Figure 3.3(a) and Figure 3.3(b) show how straight line code is transformed into SSA form. Figure 3.4(a) and Figure 3.4(b) show how two definitions are merged.

(a)	(b)

int x;
 	
x = 5;
x = 7;
x = x + 2;
	int x_1,x_2,x_3;
	
	x_1 = 5;
	S27: x_2 = 7;
	S28: x_3 = x_2 + 2;

Figure 3.3: Example transformation of straight line code into SSA form. (a) original code, (b) code in SSA form.

```
int x,y;

x = 5;
if( ... )
{
  x = x + 1;
}
else
{
  x = x + 2;
}

y = x;

        (a)
```

```
     int x_1,x_2,x_3,x_4,y_1;

     x_1 = 5;
     if( ... )
     {
       x_2 = x_1 + 1;
     }
     else
     {
       x_3 = x_1 + 2;
     }
S29: x_4 = φ(x_2,x_3);
     y_1 = x_4;

        (b)
```

Figure 3.4: Example transformation of code with `if` conditional into SSA form. (a) original code, (b) code in SSA form.

Notice that a simple way of renaming variables is to append an underscore followed by a "version" number. The number changes as a new versions of the variable is needed. An example of this occurrence is at statements *S27* and *S28* for variable `x` in Figure 3.3(b).

The second part of SSA form is the introduction of ϕ-nodes. In Figure 3.4(b) a ϕ-node is inserted at statement *S29*. The meaning of $\phi(x_2, x_3)$ is as follows:

> If flow of control reaches $\phi(v_1, v_2)$ from the left (then part) then the value of v_1 is returned. On the other hand if flow of control reaches the ϕ-node from the right (else part) then the value of v_2 is returned[1].

Assuming that the input program is in SSA form, defining the symbol to be represented by the variable name, is sufficient to overcome the problem introduced in Figure 3.2. Figure 3.5 shows the SSA form of the program segment from Figure 3.2. The problem is overcome because the occurrence of `x` in `expr2` is renamed to `x_2` at statement *S30*. This is different from `x_1` occurring in `expr1`. The following comparison test could then be used:

```
      ...
      expr1_1 = x_1 + y_1;
      ...
S30:  x_2 = ...;
      ...
      expr2_1 = x_2 + y_1 + 1;
      ...
      A[expr1_1] = A[expr2_1];
```

Figure 3.5: Program from Figure 3.2 in SSA form.

```
x_1 + y_1 ?= x_2 + y_1 + 1
```

This test would correctly answer "unknown". All would be fine at this point if it weren't for the existence of multi-level pointer languages such as C.

3.3 Extended SSA Numbering

Although SSA form has become very popular in FORTRAN compilers, it has not enjoyed such popularity among languages with multi-level pointers, such as C. I propose to introduce the properties of SSA form into compilers for languages with multi-level pointers, using a new dataflow analysis which I call *SSA Numbering*.

[1]The exact meaning of left and right as well as the order of elements in ϕ-nodes following `switch` statements is a matter of convention.

To motivate the introduction of SSA numbering instead of SSA form, let's consider an example of transforming a program into SSA form. Doing this requires two basic steps. The first inserts ϕ-nodes that join two possible values from different flow-of-control paths, and the second renames variables such that each name has exactly one definition. It is the first step that causes the most difficulty in the presence of pointers. I propose to concentrate on the second step, renaming variables by appending numbers.

Consider the code fragment example shown in Figure 3.6. Figure 3.6(a) presents the original program, Figure 3.6(b) presents the program in SSA form, and Figure 3.6(c) contains it with SSA Numbers. Notice several key differences between the original program and the program in SSA form. First, all variables are renamed, and some variables, such as `a`, which had several static definition points, became several variables; (`a_1`, `a_2`, `a_3`, `a_4`), one for each definition point. Note that separate memory locations are associated with these variables. Second, a new statement is introduced, labeled *S32*. This statement, known as a ϕ-node or join node, ensures that data correctly flows when two flow-of-control paths join. Depending from which branch the flow of control came to the ϕ-node, the corresponding value is returned.

int main()		**int** main()		**int** main()
{		{		{
int a,b,c;		**int** a_1,b_1,c_1;		**int** a,b,c;
		int a_2,a_3,a_4;		
a = 1;		a_1 = 1;		a_1 = 1;
b = 2;		b_1 = 2;		b_1 = 2;
if(a < 10)		**if**(a_1 < 10)		**if**(a_1 < 10)
{		{		{
a = a + 1;		a_2 = a_1 + 1;		a_2 = a_1 + 1;
}		}		}
else		**else**		**else**
{		{		{
a = a + b;	*S31:*	a_3 = a_1 + b_1;	*S33:*	a_3 = a_1 + b_1;
}		}		}
	S32:	a_4 = ϕ(a_2,a_3);		
c = a * b;		c_1 = a_4 * b_1;		c_1 = a_4 * b_1;
}		}		}
(a)		(b)		(c)

Figure 3.6: Example of SSA form vs. SSA Numbering.

Now let us compare SSA form to the proposed SSA numbering in Figure 3.6(c).

Notice that the SSA numbered program does not add new variables. In the example, there is only one location for the variable `a`, whereas in SSA form there were four. Also, the SSA numbered program does not have ϕ-nodes. With the exception of subscripts on all variables, the program looks exactly like the original. The subscripts are the same numbers as the ones appended to variables in the SSA form program (Figure 3.6(b)). For example, in statement *S33* and its counterpart *S31*, the left-hand sides are `a_3` and $\mathtt{a}_3$. Similarly, on the right-hand side `a_1` vs. $\mathtt{a}_1$, and `b_1` vs. $\mathtt{b}_1$. These subscript numbers are not part of the variable; instead, just like information from a program analysis, which is stored at statement points, the SSA numbers are stored at each variable reference, to be used by compiler components that wish to access them.

3.3.1 Difficulties with pointers

Looking at the differences between SSA form and SSA numbering, it is clear that there are fewer properties in the latter. SSA numbering still has the property that two variable references with the same SSA number must denote the same value. However, unlike SSA form, SSA numbers do not provide a single definition point for each use. Then, why propose SSA numbering ? The answer lies in three major problems that are easily solved in SSA numbering, two of which are illustrated in Figure 3.7.

int a;
 	
a = 1;
if(a < 10)
{
a = 2;
S34: p = &a;
}
(a)
	int a_1,a_2,...;
	
	a_1 = 1;
	if(a_1 < 10)
	{
	a_2 = 2;
	S35: p_2 = &a_?;
	}
	(b)
		int a;
		
		a_1 = 1;
		if(a_1 < 10)
		{
		a_2 = 2;
		S36: p_2 = &a;
		}
		(c)

/* *assume p points to a or b* */
c = a + b;
S37: *p = 5;
 	
d = a + b;
(d)
	c_1 = a_1 + b_1;
	S38: *p_1 = 5;
	???
	S39: d_1 = a_1 + b_1;
	(e)
		S40: $c_1 = a_1 + b_1$;
		$*p_1 = 5$;
		
		S41: $d_1 = a_2 + b_2$;
		(f)

Figure 3.7: Difficulties with pointer for SSA form shown in the original program, SSA form and SSA Numbered program.

First, consider the problem of taking the address of variables. This problem is illustrated in parts (a), (b), and (c) of Figure 3.7. Look at statement *S34*, which takes the address of the variable `a`. In SSA form, the variable `a` is represented by several memory locations, one for each static definition. Which location is being asked for? In the corresponding statement in the SSA form, *S35*, should one take the address of `a_1`, `a_2`, or maybe even something else? In the SSA Numbered program, statement *S36* does not have this problem because the variable `a` remains represented by only a single location.

Next, consider the problem of an assignment via a pointer, illustrated in parts (d), (e), and (f) of Figure 3.7. The original statement is labeled *S37*. For the purpose of the example, let us assume that the pointer `p` can point to either `a` or `b`. When attempting to put the program segment into SSA form, it is noticed that statement *S38* has an implicit write to either `a` or `b`, but it is not known which until run time. For the program to be in SSA form, it is necessary to somehow expose this implicit write. Cytron and Gershbein [CG93] have done some work to correctly model this, such that statement *S39* uses the correct variables with the correct values; however, as can be seen in Figure 3.7(f), such a construct is easily modeled in SSA numbering. The SSA number of `a` and `b` in statement *S41* are different from those in statement *S40*, to reflect that they have a new possible static definition point.

The third problem is the lack of information about the value represented by `*p`. For two scalar references with the same name, it is possible to say that they share the same static definition; however, for two references of `*p` one can only say that they share the same static definition of `p` (i.e. the address that `p` holds), and nothing about the value returned by `*p`. Extended SSA numbering will solve this.

3.4 Definition and Implementation of Extended SSA Numbering

In this section, I describe the Extended SSA Numbering analysis, by first defining the basic idea behind it and then presenting the algorithm for computing it.

The goal of SSA numbering analysis is to associate with each variable reference a number such that if the numbers were used as a renaming method the program would be in SSA form with all the ϕ-nodes removed.

Note that the major differences in generating SSA numbering and SSA form are:

- No ϕ-nodes are inserted for SSA numbering, whereas in SSA form, ϕ-nodes are a crucial part of the intermediate representation.

- When performing the renaming in SSA numbering, numbers are associated with variable references but no variables are actually renamed or created.

Let's take a closer look at what pointer variables represent and what data can be reached via a pointer. For example, with a pointer `p`, one can either use it directly (`q = p;`) or dereference it (`x = *p;`). Each method gives a different piece of data. Figure 3.8 shows how pointers are commonly drawn. It points out that there are two items of data accessible via the pointer, `p` and `*p`.

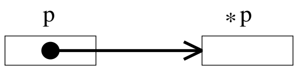


Figure 3.8: Data accessible via a pointer.

Given that pointers represent two items of data, modeling their behavior with one number will not be sufficient. I therefore introduce a second SSA number for pointer variables. I call these two numbers *primary* and *secondary* SSA numbers. Recall from chapter 2 that one of the properties of SIMPLE is that multi-level pointer references are simplified to several single-level dereferences. Due to this simplification no further SSA numbers are required for multi-level pointers.

The primary SSA number is used to represent the address the pointer holds (the data labeled `p` in Figure 3.8), and the secondary SSA number to represent the data the pointer points to (the data labeled `*p` in Figure 3.8). Therefore, if the value of the pointer is defined, a new primary SSA number is generated; and if an assignment via `*p` or via a variable `v` that `p` can point to is encountered, a new secondary SSA number for `p` is generated.

For an example of how secondary SSA numbers work, see Figure 3.9. In this example, the statements of the form `x = *p + n;` (statements *S42*, *S43*, *S45*, *S46*, *S48*, *S49*) are used to let us see the primary and secondary numbers of `p`. Take, for example, statements *S43* and *S45*. Both these statements reference `*p`. The primary numbers in both references of `*p` are the same, because they both have the same static definition for `p`. On the other hand, because there is a write to `n` at statement *S44* and `p` points to `n`, the use of `*p` at statement *S45* has a different static definition point from that of statement *S43*, thus producing a new secondary SSA number. Statement *S47* has a similar effect on references of `*p` in statements *S46* and *S48*. In addition to assignment statements, one must perform correct merging of control flow paths. Statement *S49* shows the new primary and secondary SSA numbers for `p` after a merge in control flow paths. One can imagine that there exists a ϕ-node just after the `if` statement, which would look like `*p`$_{1,4}$ `=` ϕ`(*p`$_{1,2}$`,*p`$_{1,3}$`)`.

```
        int main()
        {
          int a,x,n,*p;
          p1,1 = &n;
S42:      a1 = *p1,1 + n1;
          if (a1 <= 9)
          {
S43:        x1 = *p1,1 + n1;
S44:        n2 = x1;
S45:        x2 = *p1,2 + n2;
          }
          else
          {
            x3 = n1;
S46:        x4 = *p1,1 + n1;
S47:        *p1,3 = x4;
S48:        x5 = *p1,3 + n4;
          }
S49:      x6 = *p1,4 + n5;
        }
```

Figure 3.9: Sample program Extended SSA Numbers.

3.4.1 Implementation

With the goal of extended SSA numbering analysis in mind, I now describe the implementation of this analysis. To correctly SSA-number all variable references, one must know where ϕ-nodes would have to be inserted if one were putting the input program into SSA form. As noted in chapter 2, McCAT has a restructuring phase, which guarantees the input program will be well structured (i.e. no `goto`'s). With this in mind, I refer the reader to the paper by Brandis and Mössenböck [BM94] for a description of the placement of ϕ-nodes in structured programs for SSA form.

Figure 3.10 shows the steps that are taken for handling some of the key types of assignment statements, `if` statement, `for` loops, `while` loop, `do-while` loops, and function calls.

To assign SSA numbers to variable references, a look-up table is maintained containing the pairs (`v`, `n`) where `v` is a variable and `n` is the current SSA number assigned to the variable `v`. The table is indexed by `v`. Each time an assignment to a variable `v` is encountered or a ϕ-node for `v` is expected, the pair (`v`, `n`) is replaced by (`v`, `n`$'$), where `n`$'$ is an SSA number that has never appeared with the variable `v`. Selecting `n`$'$ is easily accomplished by having a separate master table: each time a new number is needed for `v`, its SSA number is incremented by one in the master table. The function `new_SSA(v,t)` generates a new primary SSA number for the variable `v`, and records that new number in the table `t`. For pointer variables, secondary numbers are managed similarly using the function `new_secondary_SSA(v,t)`. The function `Store_SSA(v,t,s)` is used to look up the primary and secondary numbers of `v` in table `t`, and store them in the reference of `v` at the statement `s`.

The algorithm makes use of a procedure `update_written()`, defined in Figure 3.10. This procedure handles the generation of new SSA numbers for those variables that need it. For example, a simple assignment into `a` will increment the primary SSA number of `a` (`a` is in the write set) as well as any secondary SSA numbers of pointers that can point to `a`. For an assignment into `*a`, the write set contains all the variables that `a` can point to. The procedure `update_written()` will thus generate new primary SSA numbers for these variables, and will generate new secondary SSA numbers for all pointers that can point to them, including `a`.

Note that even in the case of loops, the analysis is single-pass. This is possible thanks to the availability of write sets. Similarly, because read/write sets is an interprocedural analysis, write sets can be used to accurately model the effect of function calls to SSA numbers.

Also, notice in the algorithm that when processing conditionals a copy of `in_table` must be made. At the end of processing both sides of the conditional, the copy can be discarded. Instead of merging the two tables, the write set can be consulted to know which variables would have to be merged, and thus generate new SSA numbers for them.

3.4.2 Points of attention

There are a few additional details in extended SSA numbering that must be addressed.

First, let's consider `break` and `continue` statements. Figure 3.11(a) shows a sample while loop, and Figure 3.11(b) shows the control flow graph in SSA form, that would be built for this loop. Figure 3.11(c) shows the same while loop, but introduces a `break` statement, and Figure 3.11 is its control flow graph. Notice that at statement *S50*, it is necessary to join two values of `x`, one from outside the loop and one from previous iteration. This shows up as `x2 = `ϕ`(x1,x4);` in the control

```
fun update_written(stmt,in_table)
{
  foreach v in WriteSet(stmt)
    new_SSA(v,in_table)
    foreach pointer p
      if(at stmt, p points to v)
        new_secondary_SSA(p,in_table)
}

fun process_statement(stmt,in_table)
{
  switch(stmt)
    < a = b op c >
      Store_SSA(b,in_table,stmt)
      Store_SSA(c,in_table,stmt)
      update_written(stmt,in_table)
      Store_SSA(a,in_table,stmt)

    < a = *b >
      Store_SSA(b,in_table,stmt)
      update_written(stmt,in_table)
      Store_SSA(a,in_table,stmt)

    < *a = b >
      Store_SSA(b,in_table,stmt)
      update_written(stmt,in_table)
      Store_SSA(a,in_table,stmt)

    < if(a op b) { then_blk } else { else_blk } >
      Store_SSA(a,in_table,stmt)
      Store_SSA(b,in_table,stmt)
      in_table2 = duplicate(in_table)
      process_statement(then_blk,in_table)
      process_statement(else_blk,in_table2)
      update_written(stmt,in_table)

    < for(INIT;a op b;INCR) { body } >
      process_statement(INIT,in_table)
      update_written(stmt,in_table)
      Store_SSA(a,in_table,stmt)
      Store_SSA(b,in_table,stmt)
      process_statement(body,in_table)
      process_statement(INCR,in_table)
      update_written(stmt,in_table)

    < while(a op b) { body } >
      update_written(stmt,in_table)
      Store_SSA(a,in_table,stmt)
      Store_SSA(b,in_table,stmt)
      process_statement(body,in_table)
      update_written(stmt,in_table)

    < do { body } while(a op b) >
      update_written(stmt,in_table)
      process_statement(body,in_table)
      Store_SSA(a,in_table,stmt)
      Store_SSA(b,in_table,stmt)

    < f(a,b,...) >
      foreach function argument v
        Store_SSA(v,in_table,stmt)
      update_written(stmt,in_table)

    ...

  process_statement(NEXT(stmt),in_table)
}
```

Figure 3.10: Partial algorithm for SSA numbering analysis.

flow graph. To help distinguish variable uses inside a loop from after the loop, it was decided to insert ϕ-nodes to join the value generated after the loop finished, with the value before the loop started (in the case that the loop does zero iteration). The insertion of the join node happens at statement point *S51*. The effect of inserting a `break` statement, is that it introduces a new possible path to the end of the loop.

Adding this path introduces another argument to the ϕ-node, changing it to `x5 =` ϕ`(x1,x2,x3)` from `x5 =` ϕ`(x1,x2)`. Because extended SSA numbering concentrates on the SSA numbers, in this case 5, extended SSA numbering will not change.

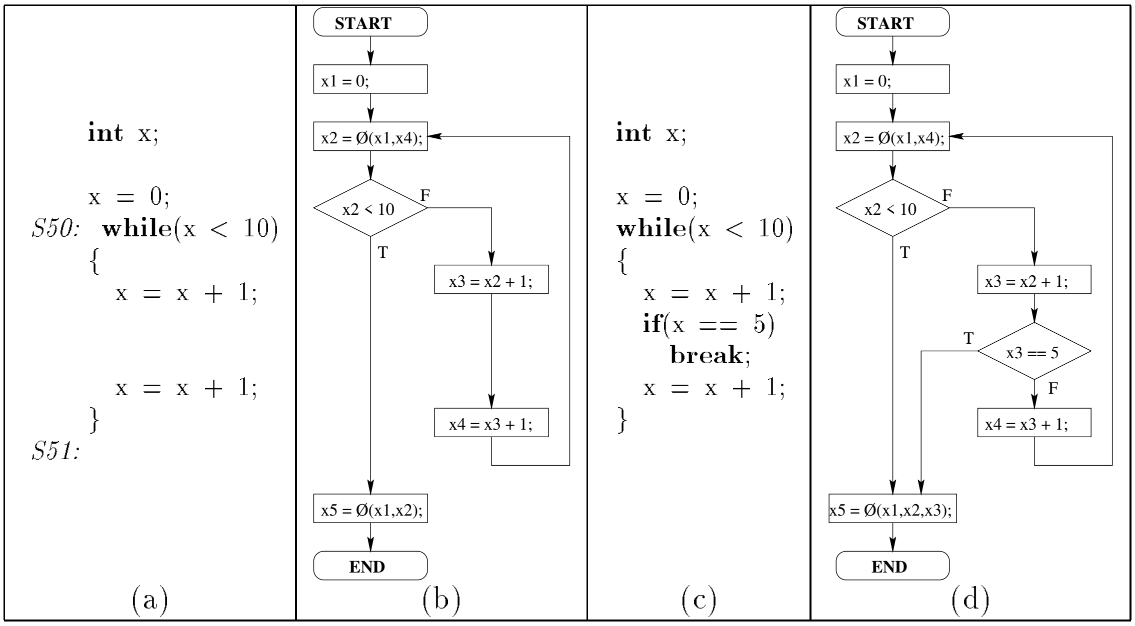


Figure 3.11: Effects of `break` statements in while loops.

On the other hand, consider a do-while loop. Figure 3.12(a) shows a sample do-while loop, and Figure 3.12(b) shows the control flow graph in SSA form that would be built for this loop. Figure 3.12(c) shows the same do-while loop, but introduces a `break` statement, and Figure 3.12 is its control flow graph. Similarly to a while loop, there is a need to join data coming from outside of the loop with data coming from the previous iteration, at statement point *S52*. As opposed to a while loop, a do-while loops execute at least once. It is therefore not needed to join data from before the loop with the data of the last iteration at the statement point *S53*; however, if a `break` statement is added a short cut is generated to the end of the loop requiring a join of data. This join appears in Figure 3.12(d) as `x5 =` ϕ`(x4,x3)`.

Now that I have pointed out the need for extra ϕ-nodes, it is necessary to ask, which variables need ϕ-nodes (or in extended SSA numbering case, which variables need a new SSA number). The answer is those variables that are are modified "after"[2]

[2]The use of the word "after" should be taken to mean any statements that follow the last branch

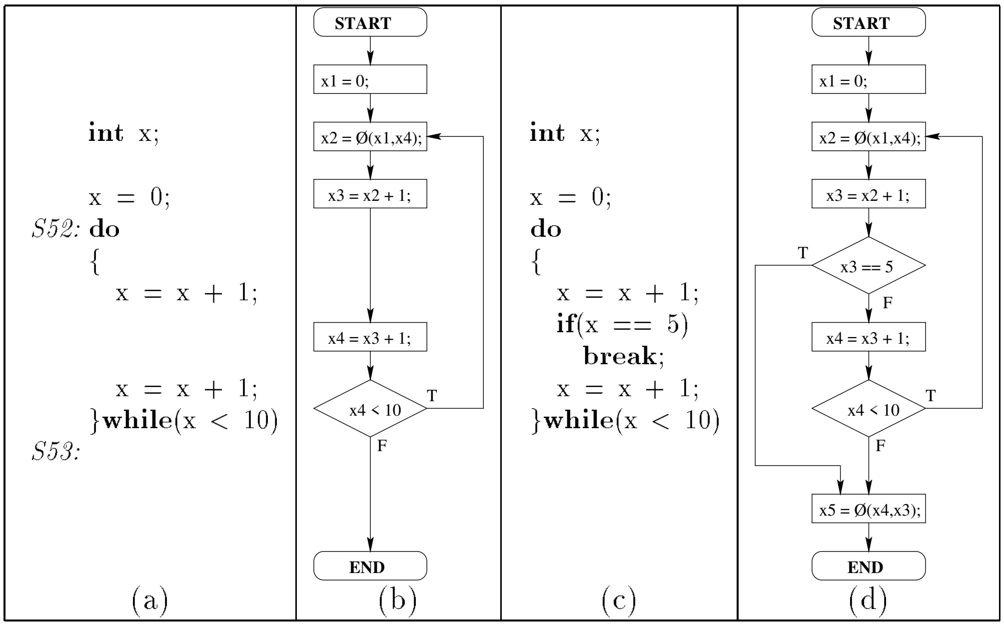


Figure 3.12: Effects of `break` statements in do-while loops.

the `break` statement. This can be done by checking the write sets of these statements and save some work using the hierarchical properties of the read/write sets. I chose a very simple conservative approach to generate new SSA numbers for all variables that are written by the whole loop.

The algorithm for do-while loops now changes in the following way. A new function `process_statement()` is added which returns `TRUE` or `FALSE` to signify if there exists a `break` statement in the passed statement block, such that it can effect a loop containing this block. For example, the passed statement block S may contain a loop L. Loop L can have a `break` statement, but this `break` statement will not influence the control flow of anything outside of S; therefore, `process_statement()` will return `FALSE`. On the other hand if there is no loop or `switch` statement surrounding a `break` statement in S then `process_statement()` must return `TRUE`.

The code for processing do-while loops is now replaced by

< **do** { body } **while**(a op b) >
  update_written(stmt,in_table)
  broken = process_statement(body,in_table)

---

containing the `break` statement. This is because by definition any statement literally after a `break` are dead code.

```
Store_SSA(a,in_table,stmt)
Store_SSA(b,in_table,stmt)
if (broken)
  update_written(stmt,in_table)
```

The only change is the additional `update_written()` call which generates new extended SSA numbers if they are needed. One must also do a very similar change for continue statements. A continue statement brings the control flow to just before the condition test. One would therefore insert the call to `update_written()` just before saving the SSA numbers of the variables in the condition.

Next, let us consider handling arrays in extended SSA numbering. Arrays can be treated as one entity. A write or read from different elements of the array, need not be distinguished. Such methods are common (see [CFR+91, pages 460-461]) for handling arrays in SSA form as well as other data-flow analyses. Each update of an array element and each access or an array element are modeled using `A = update(A,ele_num,val);` and `x = access(A,ele_num);` functions. These functions hide partial access or update of the array and treat the array as one entity.

Lastly, I must address the issue how to SSA number structures and their fields. There are two approaches to doing this.

Method 1: Only scalar and array structure fields (ie. leaf fields) are assigned SSA numbers. In the case of structure copy assignments, new SSA numbers must be generated for all such fields. For example a field reference may look like $\texttt{x = a.b.c.d}_5\texttt{;}$ assuming that the field `d` is not of structure type.

Method 2: SSA numbers are associated with all the fields as well as with the structure itself. Assignment to a field deals with the SSA number associated with that field, and a structure copy assignment changes only the SSA number associated with the structure as a whole. For example a field reference may look like $\texttt{x = a}_1\texttt{.b}_1\texttt{.c}_2\texttt{.d}_5\texttt{;}$. In this case, the meaning of two references to having the same extended SSA numbers means that all corresponding structure/field have matching extended SSA numbers.

I chose to implement method 2. Method 2 is more cumbersome but for our research team it is important to have the primary SSA number for pointer variables to structures (eg. $\texttt{x = (* p}_{1,3}\texttt{).b}_4\texttt{;}$). Figure 3.13 shows a sample program manipulating points with SSA numbers assigned to structures for both methods. In method 1, at statement *S54* new SSA numbers for both fields `x` and `y` were generated. In method

2 at statement *S55* only the SSA number for the structure a was generated, and the SSA numbers for its fields remained the same.

struct { int x,y; } a,b; **int** v; $a.x_1$ = 0; $a.y_1$ = 0; $b.x_1$ = 1; $b.y_1$ = 1; *S54:* a = b; $v_1 = a.x_2 + a.y_2$; (Method 1)	**struct** { int x,y; } a,b; **int** v; $a_1.x_1$ = 0; $a_1.y_1$ = 0; $b_1.x_1$ = 1; $b_1.y_1$ = 1; *S55:* a = b; $v_1 = a_2.x_1 + a_2.y_1$; (Method 2)

Figure 3.13: Example program showing SSA numbers for structures using both methods.

3.5 Properties of Extended SSA Numbering

As mentioned in section 3.3, extended SSA numbering is not SSA form, and does not have all its properties. Some properties were lost, and few properties for pointer variables have been introduced. Let us consider more closely the properties of extended SSA numbering.

3.5.1 Factored use-def and def-use chains

First, because extended SSA numbering has no ϕ-nodes, we cannot use it to get use-def and def-use chains. For this reason we need reaching definitions as a separate analysis. McCAT is equipped with this analysis (section 2.1.2), where for each definition point a set of statements is stored, indicating all the statements where this newly generated value may possibly be used. Also, for all variable references, a set of statements listing the possible definitions points that may reach the reference is stored at the reference point.

It is commonly pointed out that using SSA form to provide use-def and def-use chains is space-efficient, whereas the space requirement for storing sets of possible uses and sets of possible definitions can grow very quickly. In theory this is true; however, I have tested several benchmarks, and on average, programs do not require more than two or three elements per set. Considering that in SSA form one must insert ϕ-nodes, I feel the cost to be not significant.

Table 3.1 gives a short description of all benchmarks I measured as well as an idea of the size of all benchmarks by giving the number of SIMPLE statements used to represent it. Table 3.2 shows the average size of DU and UD chains of the benchmarks in Table 3.1. In each benchmark the count of elements of the reaching definitions set for each variable appearing on the right hand side of every assignment statement was made. Also for every assignment statement the number of elements in the uses set for the variable appearing on the left hand side were counted. Columns labeled *Avg # Defs per set* and *Avg # Uses per set* give the average size of the reaching definitions/uses set stored at each variable reference. Columns labeled *Sample size* state how many variables were checked to get the average sizes.

3.5.2 Definition point properties

Based on the "one definition per name" property of SSA form, we can derive the following two properties for extended SSA numbering about the value returned by a variable or indirect reference:

Property 1 *Two references of variable* `v` *at statement* S *and statement* T *will produce the same value if SSA#(*`v`*,*S*) = SSA#(*`v`*,*T*).* [3]

Note that SSA#(`v`,*S*) returns the primary SSA number of `v` at statement *S*. Similarly for indirect references:

Property 2 *Two indirect references* `*p` *at statement* S *and statement* T *will produce the same value if SSA#(*`p`*,*S*) = SSA#(*`p`*,*T*) and ExtSSA#(*`p`*,*S*) = ExtSSA#(*`p`*,*T*).* [3]

Note that ExtSSA#(`v`,*S*) returns the secondary SSA number of `v` at statement *S*.

Further, using extended SSA numbers as part of the name we can derive two more properties:

Property 3 *Two references of variable* `v` *at statement* S *and statement* T *will have identical sets of possible definition points if SSA#(*`v`*,*S*) = SSA#(*`v`*,*T*)*

[3] *Because both extended SSA numbering and SSA form are static analyses, it is valid to assume statements are within the same loop iteration and function call invocation.*

Benchmark Name	Description	# SIMPLE stmts
`spanning`	Calculate minimum spanning circle for set of points	271
`sort`	Bubble sort, Quick sort, Quick sort 3-median	145
`determ`	Compute determinant of matrix	129
`knight`	Knight's Tour problem	103
`complex`	Find complex roots of polynomial	352
`heap`	Heap sort	138
`mersenne`	Computes digits Mersenne prime $M = (2^p - 1)$	134
`sort2`	Different sorting algorithms	114
`cluster`	Graph clustering algorithms	706
`travel`	Traveling salesman problem	450
`frac`	Represent floating point as quotient of 2 integers	126
`exptree`	Build expression tree out of set of integers and operators to evaluate to given total	289
`nsieve`	Sieve of Eratosthenes	182
`paging`	Virtual memory simulation	276
`nqueens`	N Queens problem	333
`sim`	A time-efficient, linear-space local similarity algorithm	1821
`unroll`	Matrix multiply with 0 to 8 iterations unrolled	486
`tomcatv`	Mesh generation with Thompsons solver, vectorized version	608
`nrcode2`	One part of application suite for vectorizing C compilers	232
`asuite`	Test suite for vectorizing C compilers (Argonne)	449
`pgm4`	Hash table implementation with linked list buckets	114
`pgm6`	Construct binary trees as indexed data structure	140
`dhrystone`	Dhrystone benchmark	258
`intmm`	Integer matrix multiply	54
`polymesh`	Compute normals for a polygon mesh at each vertex	339
`grid`	Simple average processing on all matrix elements	208
`lorenz`	Solves the "Lorenz" equations using Runge-Kutta 4	154
`matrix`	Matrix multiplication	803
`matrix2`	BalanceMatrixt, ElmHes, Four1 from Numerical Recipes	500
`newton`	Newton method for finding the sqrt(2.01) given sqrt(2.0), sqrt(2.1), sqrt(2.2), ...	84
`genetic`	Genetic program, population size etc...	388

Table 3.1: Description of benchmarks used to measure space requirements of DU, UD chains.

Benchmark	Avg # Defs per set	Sample size	Avg # Uses per set	Sample size
spanning	1.09	249	1.32	222
sort	1.25	59	1.88	92
determ	1.57	77	1.76	88
knight	1.56	48	1.90	58
complex	1.17	225	1.48	252
heap	1.53	72	1.84	98
mersenne	1.54	48	1.71	80
sort2	1.76	45	2.37	70
cluster	1.41	239	2.41	421
travel	1.64	210	1.85	313
frac	1.61	84	2.01	76
exptree	1.50	123	1.62	167
nsieve	2.19	78	2.07	97
paging	2.21	53	2.50	143
nqueens	4.81	142	1.57	205
sim	2.53	1180	3.14	1194
unroll	1.50	265	2.20	346
tomcatv	1.36	567	1.85	523
nrcode2	3.20	133	2.54	163
asuite	1.51	145	2.41	252
pgm4	1.55	33	1.57	70
pgm6	1.82	67	1.61	76
dhrystone	1.30	134	1.43	144
intmm	1.42	19	1.82	33
polymesh	1.68	117	1.31	210
grid	1.28	163	1.81	178
lorenz	3.13	108	2.19	93
matrix	1.52	461	2.59	552
matrix2	1.74	452	2.33	398
newton	1.37	41	1.96	55
genetic	2.07	129	1.86	217

Table 3.2: Average sizes of defs and uses sets stored at each variable reference.

Property 4 *Two indirect references* `*p` *at statement* S *and statement* T *will have identical sets of possible definition points if SSA#(*`p`*,*S*) = SSA#(*`p`*,*T*) and ExtSSA#(*`p`*,*S*) = ExtSSA#(*`p`*,*T*).*

It was the intention in designing this analysis to make these properties available to optimizing transformations and other analyses.

3.6 Defining "symbol"

I conclude this chapter by defining "symbol" for the remainder of this thesis.

Definition 1 *A* symbol*, for some reference, is composed of a variable name, optional dereference operator and the variable's extended SSA numbers. In the case of a structure field reference, all the names with their extended SSA numbers that appear in the reference are part of the* symbol

With this definition, and in the context of basic blocks with conditional statements, two references to the same symbol are guaranteed to give the same value. This comes from properties 1 and 2 of extended SSA numbering. The only constraint that remains is cross-loop-iterations equivalence. This equivalence was also not supplied by SSA form and in general it cannot be provided with a static analysis. Dynamic analysis requires running the input program which is not an available option. This limitation of cross-loop-iterations equivalence, must be kept in mind through out the entire design of the symbolic dependence analysis framework.

Table 3.3 gives some example symbols that may be generated in McCAT under the symbolic array dependence analysis framework. Note that the difference between a dereferenced pointer and non-dereferenced pointer is preserved by the presence or lack of the dereferencing operator ($*$ in C). For the case of a plain pointer reference, the secondary SSA number has no effect on the equivalence of references. However, this does not come into play in array dependence analysis because all symbolic expressions come from index expressions that are of integer type.

$\texttt{x}_5$	Scalar variable
$\texttt{*p}_{1,2}$	Dereferenced pointer
$\texttt{p}_{1,2}$	Non-Dereferenced pointer (Only primary SSA Number applies)
$\texttt{s}_2\texttt{.t}_4$	Field reference from structure
$\texttt{(*sp}_{2,3}\texttt{).t}_4$	Field reference from pointer to structure

Table 3.3: Example symbols that can be encountered in McCAT symbolic array dependence analysis framework.

Chapter 4

Index Expression Analysis

In this chapter, the main goal is evolving a process to describe the value represented by an expression. This is trivial at run time, but when compiling the program, the value of an expression is usually dependent on input not yet available. One could simply state that each expression in the program possibly has different value, but then comparison of expressions would not be effective.

In the framework of symbolic array dependence testing, *Index Expression Analysis* summarizes each array index expression for effective comparison. Its placement in the framework as well as its components are show in Figure 4.1.

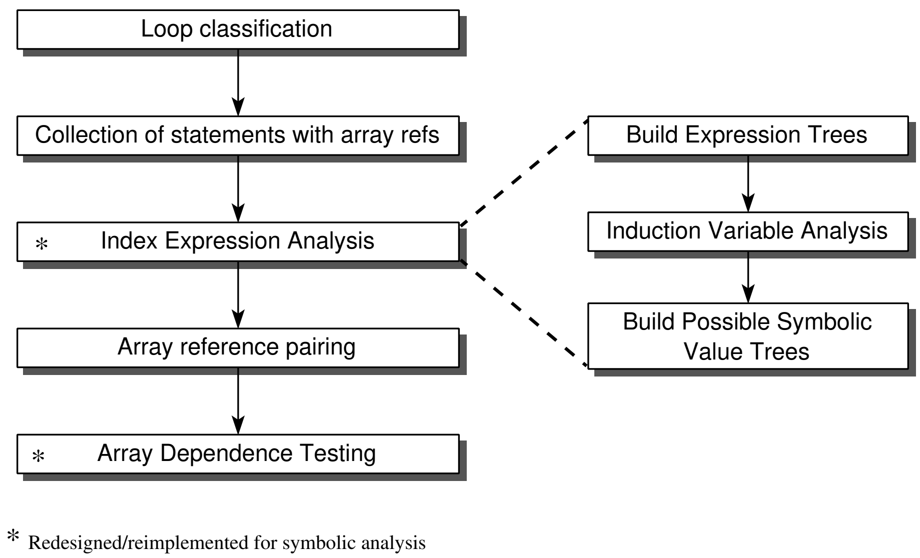


Figure 4.1: Index expression analysis in symbolic array dependence framework.

4.1 Motivating Example

As mentioned earlier, the goal is to represent an expression so that it can be effectively compared to other expressions. Such comparisons of expressions will be typically used for comparison of array index expressions. Consider an example of four array references as shown in Figure 4.2.

```
        void f()
        {
          ...
          v = w;
          if( ... )
          {
            x = p;
          }
          else
          {
            x = r;
          }
          t = u + v;
          s = t;
          for( ... )
          {
S56:        A[s] = 3;
S57:        A[t] = 4;
S58:        A[x] = 5;
            temp = x + 1;
S59:        A[temp] = 6;
            ...
          }
        }
```

Figure 4.2: Motivating example for index expression analysis.

First consider the index expression in statement *S56* (`s`). The value of `s` is unknown at compile time. It is possible to treat it as a symbolic unknown just like in mathematical equations. It may be possible to compare it to another index expression such as `s`, or even `s + 1`, and come up with useful information like “equal” or “different by 1”, respectively. Unfortunately, it will not be possible to compare it to the index expression at statement *S57* (`t`). Both `s` and `t` will be unknown symbolic values making their comparison impossible. A possible solution to this is to follow the definition of `s` and recognize that `s` is really `t`, and then the comparison will work.

The big question arises: "How far should definitions be followed?". If definitions are followed only within the loop structure being analyzed temporary variables introduced by the compiler will be resolved. This isn't enough for the example of `s` and `t` because these variables are both defined outside of the loop structure and their comparison is still not possible. Does this mean definitions should be followed as far as possible? Even past function boundaries? If definitions are followed to the top of the function, the following substitutions for `s` will be applied

$$\texttt{s} \rightarrow \texttt{t} \rightarrow \texttt{u + v} \rightarrow \texttt{u + w}$$

and a similar substitution for `t`. Both substitutions end in `u + w`. This means that the value of both `s` and `t` can be represented by `u + w`, hence it can be concluded that `s` and `t` are equal.

This seems fine, but what happens if, during the substitution of one of the variables, more then one possible definition is encountered. Should following definitions stop or keep on going considering both branches? In the example, the index expressions in statements *S58* and *S59* have such a situation.

$$\texttt{x} \begin{array}{l} \nearrow \; \texttt{p} \\ \\ \searrow \; \texttt{r} \end{array}$$

$$\texttt{temp} \rightarrow \texttt{x + 1} \begin{array}{l} \nearrow \; \texttt{p + 1} \\ \\ \searrow \; \texttt{r + 1} \end{array}$$

How to compare such index expressions? They are clearly different by 1. Section 4.4 addresses this issue.

The remainder of this chapter is organized according to the components of index expression analysis as shown in Figure 4.1. Section 4.2 covers building expression trees, section 4.3 covers induction variable analysis and sections 4.4 and 4.5 cover constructing possible symbolic value trees.

4.2 Tree Representation

The first issue of summarizing index expressions for effective comparison is how to represent simple arithmetic expressions such as `1 + x * y`, and how to represent the definitions of each of the symbol in the expression. This issue is resolved by building an expression tree (Figure 4.3).

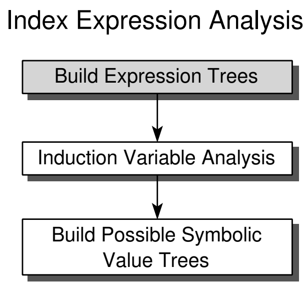


Figure 4.3: First step of induction expression analysis: Build expression trees.

Arithmetic expressions can be represented by a tree. Inner nodes are operations and the children of these nodes are the operands of the operation. Leaf nodes are either numeric constants or symbols. For example, the tree representation of the expression $1 + x \times y$ is given in Figure 4.4(a).

If the expression comes from an imperative program, it is very likely that the symbols x and y were assigned some value in a previous statement. If this statement can be found, the tree representation of this value can be attached as a subtree of the symbol node. For example, if the program segment being analyzed is such as the one given in Figure 4.4(b), a larger and more descriptive expression trees can be derived.

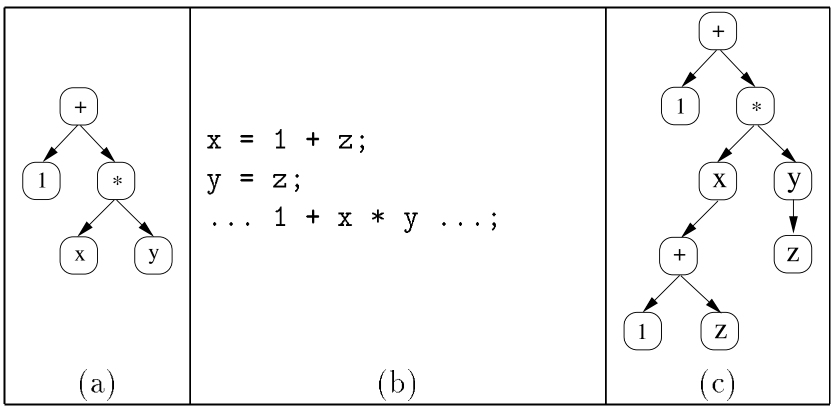


Figure 4.4: Expression tree for the expression $1 + x \times y$.

Building such a tree representation is not difficult. Using the fact that expressions in McCAT are simplified to be at most one operation makes the job even easier.

Finding the statements that define the value of a variable in the expression involves reaching definitions, a data flow analysis described in section 2.1.2. Using reaching definitions, one can ask which statement defines the value of a variable used in the expression. The expression tree for this variable can then be constructed and attached to the expression tree.

Following reaching definitions to construct a tree representing an expression is called *Backward Substitution* or *following reaching definitions.* The process is recursive in nature and it is often useful to concentrate on only a single branch of an expression tree. Instead of considering how to process an expression that contains a pointer dereference, the problem is broken down into just the situation of a single pointer dereference.

4.2.1 Possible Reaching Definitions

When following reaching definitions to build an expression tree, it was mentioned that a variable at some point in a program, may hold a value generated by one of several possible statements. This occurs in the presence of conditional statements. Depending on some condition, the variable is defined to be one value or another.

To represent such a value one could either treat the variable as a symbol with unknown value or model the two possibilities by attaching two possible sub-trees as the children of the symbol node. Figure 4.5 shows an example program segment where a variable x has two possible definitions. In this example, the variable `x` at the program point of the expression `z + x` has two possible definition points. One is the definition `x = y` and the other `x = 1 + y`. Therefore, two expression trees are built one for the expression `y` and the other for `1 + y`. Both are attached as the children of the symbol node for `x`.

4.2.2 Working With Pointers

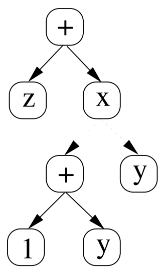
The presence of pointers tends to make analysis of C programs slightly more complex. In the situation of expression trees and the environment of McCAT, it is not very difficult.

There are two possible cases, one where the points-to relations are not known, and the other where points-to analysis give a set of named locations the pointer can point to.

In the first case, where location is not known, expanding the tree can be stopped and the dereferenced pointer can be treated as a symbol with unknown value.

```
if(...)
{
  x = y;
}
else
{
  x = 1 + y;
}
... z + x ...;
```

(a)



(b)

Figure 4.5: Expression tree with multiple possible definitions.

In the second case, where points-to analysis gives a list of variables the pointer can point to, the list of variables is treated as a list of possible definitions. For example, say that a pointer `p` is dereferenced in statement S containing the expression being analyzed. Also, suppose that points-to analysis says at program point S the pointer `p` can point to one of the following named locations `{v1,v2,...,vn}` . This list is then treated as a list of possible definitions saying that `*p` at statement S is defined at statement S to have the value `v1`, or at statement S to have the value `v2`, or etc... In the case that the named location given by points-to analysis is not desired (eg. heap, non-integer type) it is always possible to fall back to treating this particular possible value as unknown represented by the symbol `*p`.

4.3 Induction Variables

In the previous section I discussed building expression trees from expressions. The discussion is not complete without considering the effects of loops on values of expressions.

Expressions whose values are the same at each iteration of a loop nest are called *loop invariant*. These expressions present no obstacle. Following reaching definitions presents definitions always outside of the loop and therefore are treated according to the general rule.

Expressions that change value across loop iterations need special care. It is always possible to fall back on declaring a variable to be unknown symbol. Unfortunately this symbol is only constant within the same iteration of the loop. This is only useful in optimization of a basic block. The study of induction variables helps to improve the situation (Figure 4.6).

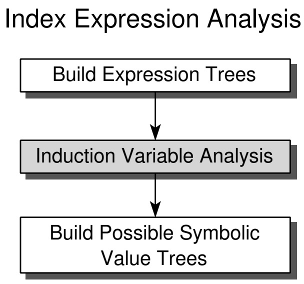


Figure 4.6: Second step of induction expression analysis: Induction variable analysis.

4.3.1 Background on Induction Variables

An *induction variable*(IV) is defined as a variable whose value is systematically incremented or decremented by a constant value in a loop [ASU86]. The example in Figure 4.7(a), a program segment, contains a variable `k`, a *basic induction variable* [Wol92]. The value of `k` changes regularly and one can write it in terms of the loop variable `i` as `2*i + 1`. It is then possible to replace the expression `k` by this value as shown in Figure 4.7(b).

```
k=0;
for(i=0;i<N;i++)
{
  ... A[k] ...
  k = k + 2;
}
```

(a)

```
k=1;
for(i=0;i<N;i++)
{
  ... A[2*i+1] ...
  k = k + 2;
}
```

(b)

Figure 4.7: Example basic induction variable and its replacement.

Details about induction variables and finding expression in terms of loop variables describing their value can be found in the works by Michael Wolfe [Wol92, GSW95].

This summary of induction variable analysis is based on the one found in Masters thesis by Justiani [Jus95].

The types of induction variables[GSW95] are as follows:

1. Basic Induction Variable (BIV): IV = IV + *constant/loop-invariant*
2. Derived Induction Variable (DIV): IV = IV_0 + *constant/loop-invariant* where IV_0 is a previously found BIV.
3. Mutually Updated Induction Variables (MUIV): A variable i is a MUIV if i occurs more then once as a DIV in the same loop body.
4. Nested Induction Variable (NIV): Either a BIV, DIV, or MUIV occurs in a perfectly-nested loop (i.e. no statements between loops in the nesting) [Jus95] with more then one nested level.
5. Multi-level Induction Variable (MIV): Either a BIV, DIV, or MUIV occurs in a nested loop (i.e. with possible statements between loops in the nesting) with more then one nested level.
6. Periodic Induction Variable (PIV): IV = *constant/loop-invariant* - IV
7. Wrap Around Induction Variable (WAIV): A variable that will become a BIV, DIV, or MUIV only after a finite number of iterations of the loop.
8. Non-Linear Induction Variable (NLIV): These can be either *geometric* or *polynomial*.
   (a) Geometric IV: IV = IV * *constant/loop-invariant*
   (b) Polynomial IV order 2: IV = IV + IV_0 + *constant/loop-invariant*
   (c) Polynomial IV order 3: IV = IV + IV_0 + IV_1 + *constant/loop-invariant*
   (d) Polynomial IV order n: IV = IV + IV_0 + IV_1 + $\cdots$ + IV_n + *constant/loop-invariant*

McADT, the array dependence tester for McCAT, had implemented the resolution of BIV, DIV, MUIV and NIV into formulae under the condition that all occurrences of *constant/loop-invariant* had to be actual integer constants or variables that can be resolved into constants using constant propagation. The following section describes how induction variable resolution was generalized and improved with the availability of symbolic framework.

```
        k1 = 0;
        for(...)
            {
S1          ...k2...
            k3 = k2 + e1;
S2          ...k3...
            k4 = k3 + e2;
            }
```

(a)

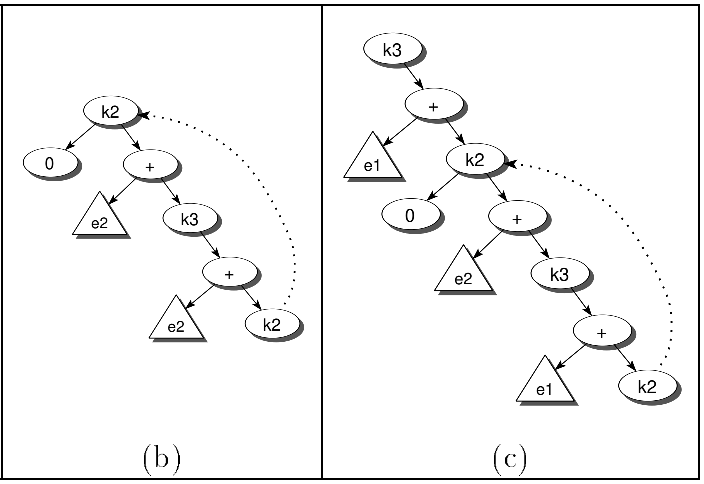


Figure 4.8: Example induction variable patterns

4.3.2 Symbolic Variation of Induction Variable Analysis

In order to discuss any symbolic variations on induction variable analysis let's look at the patterns in expression trees generated by induction variables. Consider a variable `k` that is incremented several times in a loop. It would generate an expression tree like the one in Figure 4.8(b) for the expression at statement *S1* in the program segment in Figure 4.8(a).

The pattern has a repeating node namely the one containing $\mathtt{k}_2$. It is easy to recognize such a pattern as it is being created. For any induction variable pattern there will be one node that has more then one definition (one from outside loop, and one from previous iteration). If there is more then one possible definition in any other place then the induction variable is updated under a conditional statement, or indirectly via a pointer with more then one possible points-to relation[1]. Also, one can easily ensure that the node with more then one possible definition only occurs at the top of the pattern by checking for the pattern only at nodes with more then one possible definition. This would essentially lead to "unrolling" a part of the pattern. For example the pattern would then look like the one in Figure 4.8(c) which is the tree expression for $\mathtt{k}_3$ of statement *S2* of the program in Figure 4.8(a).

This pattern is general enough to resolve BIV, DIV, MUIV, NIV and MIV. The next

[1]In situations were all paths of the conditional can be proven to increment the induction variable by the same amount, the induction variable can still be determined. This was not implemented in McCAT. Gerlek et al. [GSW95] suggested this method for improving induction variable analysis.

step is examine this pattern if it can be safely resolved into its equivalent formula. To do this the following are checked.

1. The top node of the pattern must contain only one branch inside of the loop and at least one branch to the outside. It is fairly easy to handle multiple starting values for an induction variable.

2. The top node of the pattern comes from inside the loop structure. At least one of the branches comes from outside of the loop construct, as described in step 1. It is necessary to determine which loops are involved in the IV, as well as information about their bounds. These loops are therefore checked to be admissible loops as defined by [JH94, Jus95]. The list of loops involved must also be same for each possible starting value of the IV. (i.e. can't have the IV reset in the middle of the loop nest)

3. The repeating path of the induction variable must be checked for the following

   (a) The side branches (labeled *e1* and *e2* in Figure 4.8) must be loop invariant. These can be symbols, but must be loop invariant with respect to the outer most loop involved in the IV determined in step 2.[2]

   (b) Subtract operation: Must ensure that the IV symbol appears in the left branch.

   (c) Multiply/Divide operation: Allow only multiplication and division by 1.

Additional care must be taken for a loop with an induction variable that is either incremented according to the permitted operations or not incremented at all. The conditional that implements such an induction variable does not appear in the tree pattern. Because of this, a special check had to be implemented to ensure correctness.

Building the formula

After the pattern is confirmed to be "friendly", the formula for the IV is constructed. The first step is to have an expression for the amount that the IV is incremented at each loop iteration. To do this, cut the tree component out and replace the final node in the pattern by 0. This is shown in Figure 4.9(a). Then given an iteration vector

[2]A symbol is loop invariant with respect to loop L if all the definitions of the symbol lie outside of loop L. The case such as `k = k + temp` and `temp` is defined in loop L with all of the variables appearing in the right hand side of the definition of `temp` are invariant, is also considered and recognized as invariant.

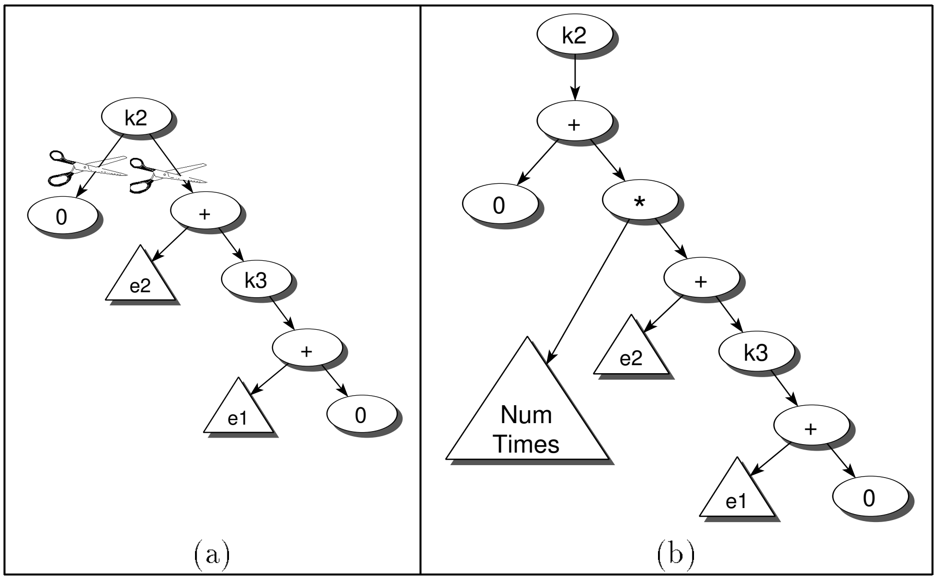


Figure 4.9: Transforming IV pattern into expression in loop variables

(loop variables), construct an expression for the current iteration count of the loop body. This involves construction of an expression tree containing loop variables and upper bounds. The last step is to multiply the repetitive addition component by the "number of times" tree and add it to the starting value of the IV. This step is shown in Figure 4.9(b).

Finished loop IV values

When building expression trees, reaching definitions are followed backward to try and find out the value of some variable. It is possible to follow a reaching definition into a previous loop. This means that the loop has finished executing and the value of any IV's will be the *finished loop* value. Figure 4.10, shows an example situation, where an array reference `a[j+k]` at statement *S60* results in following the definition of `k` into a loop that has already finished executing. Replacing the IV with its corresponding formula based on loop variables (`i`) is not desired, because the loop has already finished executing. In the example, the formula for `k` would be `2*i`. This formula is the value of `k` while the loop is executing, where as the desired value is the value of `k` after the loop finishes executing. This is easily fixed by by replacing the loop variables in the formula by their last known values plus one extra iteration. These last known values are the upper bounds of the loops involved. In the example, this results in the formula for `k` to be `2*n` because the last value of `i` was `n-1` resulting in `2*n-2` plus one more iteration.

```
        k=0;
        for(i=0;i<n;i++)
        {
          ...;
          k = k + 2;
        }
        for(j=0;j<n;j++)
        {
S60:      ... = A[k];
        }
```

Figure 4.10: Example of finished loop value for an induction variable.

Determining when to use loop variables and when to use their last known values, is an easy check testing if the statement where we started building the expression tree, is contained in the loops involved in the IV. If not then the desired value is the finished loop value of the IV, otherwise the loop is still executing and loop variables are desired.

4.4 Possible Symbolic Value Trees

Having looked at ways to expand the meaning of an expression, I now describe how to represent a value which may contain symbols in a way that facilitates comparison of two values. This can be done by building Possible Symbolic Value trees (Figure 4.11). Recall that the goal is to be able to tell if two expressions are equal or not. A natural way to accomplish this is to subtract one expression from the other taking into consideration unknown symbolic constants.

4.4.1 Sum of Products

The first step of performing a successful symbolic subtraction is to consider commutativity and distribution rules of arithmetic. The same expression can be written in

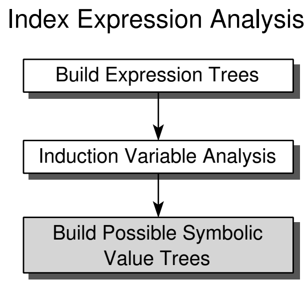


Figure 4.11: Third step of induction expression analysis: Build possible symbolic value trees.

several ways. For example the expressions

$$\begin{array}{ll} (x+y)(w+z) & \text{(Product of Sums)} \\ xw+xz+y(w+z) & \\ \left.\begin{array}{l} xw+xz+yw+yz \\ zx+yw+wx+zy \end{array}\right\} & \text{(Sum of Products)} \end{array}$$

are all equivalent yet written differently. They would also all generate different expression trees as described in section 4.2.

Due to this reason, it is necessary to agree on a standard representation. Two possible ways of standardizing expression involving only addition, subtraction and multiplication are *Sum of Products* or *Product of Sums*. Both standards are acceptable but a sum of products does not need brackets to write down because of precedence rules. Because of this reason I chose to use sum of products as the standard.

In addition to standardizing the format according to distribution rules of arithmetic, the operations of addition and multiplication are commutative. This allows us to write the symbols in any order. Lack of order will make comparison difficult. For example comparing the two terms zx and xz would be inefficient if order was not introduced. One possible order is alphabetic on the name of the symbols.

Having decided on the representation of simple expressions one can easily write an algorithm to transform an expression tree (not considering reaching definitions) into a sum of products abstract representation. Figure 4.12 shows an example expression represented textually, as an expression tree and as a sum of products.

This strategy is used by Haghighat in [Hag90]. Performing arithmetic (addition, subtraction, multiplication) on such a representation is just a programming exercise. Standardizing a representation for division is more complex, but some work on this has been also done by Haghighat [HP96] with source of [DST93]. In McCAT, only symbolic addition, subtraction, and multiplication were implemented.

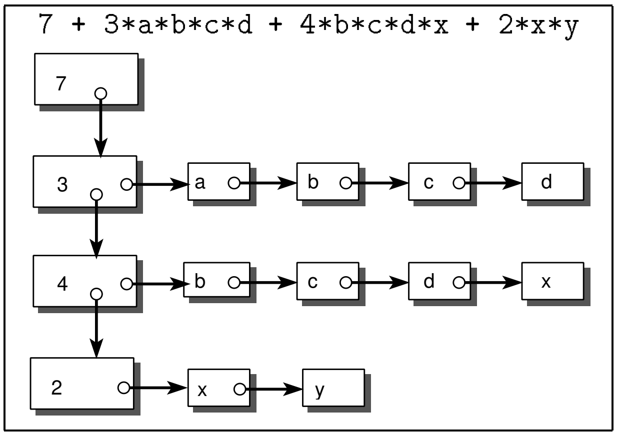


Figure 4.12: Representing Sum of Products.

4.4.2 Preserving Intermediate Values

Representing an expression as a sum of products is a good idea, but consider an expression that has several possible values, like the one in Figure 4.13. Figure 4.13(a) shows the code that generates such expressions and Figure 4.13(b) lists the possible values. The expression `e1` is either `1*x + 1*w` or `1*x + 1*z` represented as a sum of products ordered alphabetically. This seems to be not a problem, one could just represent each expression as a list of possible values. All that needs to be done is to modify the algorithm to handle symbol nodes that have children.

This turns out not to be good. Consider comparing this expression `e1` with `e2` just following it. Expression `e2` is very similar to `e1` and has possible values `1 + 1*x + 1*w` or `1 + 1*x + 1*z`. Starting to compare the expressions a attempt to symbolically compute `(1*x + 1*w) - (1 + 1*x + 1*w)` is made, coming up with the answer `-1`. Therefore the conclusion is that the two expressions are different. However, the comparison is not finished because both `e1` and `e2` can have other possible values. Figure 4.13(c) shows the tests that need to be performed. Two of them successfully show the difference to be `-1` but the other two give the answers `-1 + 1*w + -1*z` and `-1 + -1*w + 1*z`. Because `w` and `z` are unknown symbolic constants, the test fails even though the expressions are clearly different by 1.

In order to fix this situation, it is necessary to somehow preserve intermediate values. Consider the representation in Figure 4.14. This representation is called *Possible Symbolic Value trees* (PSV trees) and the structure of this representation is as follows:

`if(...)`
`  y = w;`
`else`
`  y = z;`
`v = x + y;`
`e1 = v;`
`e2 = v + 1;`
	Possible values for `e1`
	`x + w`
	`x + z`
	
	Possible values for `e2`
	`1 + x + w`
	`1 + x + z`
		Equality tests:
		`(x + w) - (1 + x + w)`
		$\Rightarrow$ `-1`
		`(x + w) - (1 + x + z)`
		$\Rightarrow$ `-1 + w - z`
		`(x + z) - (1 + x + w)`
		$\Rightarrow$ `-1 + z - w`
		`(x + z) - (1 + x + z)`
		$\Rightarrow$ `-1`

(a)	(b)	(c)

Figure 4.13: Example comparing two expressions.

- Each node of the tree contains a sum of products representation of an expression.
- The root node contains the expression as it appeared at the use point in the program without following any reaching definitions.
- A tree node has children by replacing all occurrences of one symbol in the expression by its definition. The number of children is the number of possible definitions for the symbol being replaced.
- A tree node is a leaf if all symbols in the expression it holds have unknown definitions or if it was determined to leave the symbol as an unknown symbolic constant.

Using PSV trees the intermediate values are preserved. A testing algorithm is free to inspect these values and possibly come to a better conclusion. In the example, the algorithm for testing equality can compare `v` with `1 + v` and immediately conclude a difference of 1.

4.4.3 Compressing Tree Nodes

Although the idea of PSV trees is good, it has a flaw. The flaw lies in the number of tests that must be performed in the worst case. If the PSV trees are used, it is possible that the number of required tests will grow quickly, and if array dependence tests are performed on each possible pair, efficiency may become an issue. For example, to test

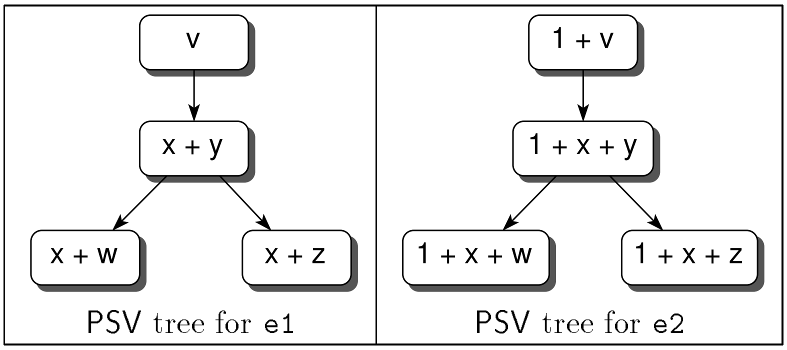


Figure 4.14: Example Possible Symbolic Value Trees.

all pairs of expressions in Figure 4.13(b) one needs only 4 tests, but to test all pairs of the PSV trees in Figure 4.14, one needs 16 tests.

To improve on efficiency, consider "compressing" the PSV tree. Compression is achieved by removing nodes with one child and connecting their child in their place. The effect will be that all nodes in the PSV tree will have either more then one child or they will be leaf nodes. Figure 4.15 shows the "compressed" PSV trees for the expressions of `e1` and `e2` from the example in section 4.4.2. Observe in Figure 4.15(a) that the intermediate node containing `v`, which had only one child, is removed and the tree root is now `x + y`. Similar compression is done for the PSV tree representing the expression `e2`. Note that one requires 16 tests to compare all pairs in the trees in Figure 4.14, but only 9 tests for Figure 4.15.

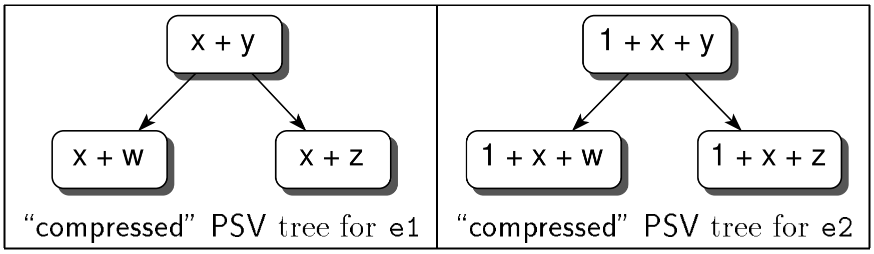


Figure 4.15: Example "Compressed" Possible Symbolic Value Trees.

To summarize the complexity issues, consider the number of possible pairing of nodes. In the case of a list of possible expressions each element of a list is a node, and in the case of PSV trees a node is a tree node.

List of possible expressions: Assuming that the number of possible expressions is n for both expressions, it is necessary to compare all possibilities of one expression with all of the other giving n^2 possible pairs.

PSV trees: To analyze the worst case scenario, let's assume that all nodes need to be compared to prove or disprove equality. The algorithm may attempt many combinations of pairs and logically combine their results, but caching the result of a pair test will at most test all possible pairs. Considering a compressed PSV tree where each node is either a leaf or an inner node with more then one child, the largest ratio of non-leaf nodes to leaf nodes, yielding the worst case, is a complete binary tree. An expression with n possible expressions will generate a PSV tree with n leaf nodes. In the worst case this will be a complete binary tree. A complete binary tree with n leaf nodes has $n-1$ inner nodes, therefore pairing $2n-1$ nodes of one tree with $2n-1$ nodes of the second tree yields $4n^2-4n+1$ pairs. Hence the number of tests will be at most 4 times the number of tests needed for the list of expressions. This was not guaranteed without "compression".

4.5 Building PSV tree algorithm

The overall algorithm for building PSV trees contains three general steps.

1. Build a basic expression tree following reaching definitions.
2. Perform improving transformations on the expression tree such as induction variable analysis/transformation.
3. Convert the expression tree into a PSV tree.

Step 1 is fairly straight forward, step 2 is highly dependent on how much effort one is willing to spend to get better expression trees, and step 3 is described in this section.

In order to support compression correctly it is necessary to give priority to symbols with only one definition. For this a priority queue is used with the operations `insert_symbol()`, `dequeue_symbol()`, `head_symbol()` as well as `copy_queue()` which makes a copy of the queue. Highest priority for dequeue is given to symbols that have only one possible definition, second priority is given to pointer dereferences which are replaced by the possible named locations they point to, and lastly to symbols with more then one possible definition.

The algorithm for building the PSV tree is given in Figure 4.16. It is started by building a single PSV node containing the start expression and the queue initialized to hold the root node of the expression tree as defined in section 4.2. The call to `expand_single_defs()` simply changes the expression held by the PSV node until it runs out of symbols with single definitions to expand. As the algorithm follows definition expressions new symbols are introduced to the queue.

```
fun buildPSV(node,queue)
    expand_single_defs(node,queue);
    s = dequeue_symbol(queue);
    for each d ∈ s.tree.children
        child_queue = copy_queue(queue);
        child = make_node();
        child.contents = expand_symbol(node.contents,s.sym,d,child_queue);
        buildPSV(child,child_queue);
        add_child(node,child);

fun expand_single_defs(node,queue)
    while head_symbol(queue) has only one possible definition
        s = dequeue_symbol(queue);
        d = s.tree.children; /* Only one child */
        node.contents = expand_symbol(node.contents,s.sym,d,queue);

fun expand_symbol(exp,sym,def,queue)
    def_exp = build_sum_of_prod(def);
    for each symbol s in def_exp
        insert_symbol(s,queue);
    return (substitute(exp,sym,def_exp));
```

Figure 4.16: Algorithm for building PSV trees

The queue is made up of pairs: the symbol, and a pointer to the node in the expression tree. Field referencing is used to access the symbol (`s.sym`) and the expression tree node (`s.tree`).

The procedure `expand_symbol()` first creates a sum of products from the expression tree. The construction of a sum of products stops at each symbol node (ie. does not follow definitions). This is achieved using the function `build_sum_of_prod()`.

The algorithm then inserts any new symbols into the queue and perform the substitution using one of the operations provided by the sum of products package that must be implemented.

The case of symbols with multiple definitions the expansion process is similar, except that instead of replacing the sum of products stored in a PSV node, a new PSV node is created using `make_node()` and attached as the child using `add_child()`. In addition to creating a new PSV node, it is necessary to preserve the queue status so that when processing the next child, the queue is intact for the next possible symbolic value.

Chapter 5

Symbolic Array Dependence Testing

With the material covered so far and the work of Justiani in [Jus95], it is possible to analyze an input program and annotate each array reference with an internal representation of its index expressions. It is now time to take two such array references in the context of a loop nest, and determine if any dependencies exist between them. As Figure 5.1 shows, before getting to the stage of actual dependence testing, the following information is available:

- Determined admissible loops and collected information about the lower and upper bounds of the loop variables from each of these loop.
- Identified statements that are relevant to Array Dependence Testing.
- Internal representations of index expressions suitable for symbolic comparisons.
- Within each loop, array references that may cause a dependence have been paired up. This was done by just considering array names and their points-to relations.

It is now time to start considering the value of the index expression to determine dependencies. The main question is "how?". The internal representation of an index expression is a PSV tree. There are many expressions stored in a PSV tree. Which one should be chosen? Perhaps a series of combinations should be used. Such questions are discussed in section 5.1. Another part of this question "how?" is: what should be done with a test pair involving symbolic unknowns? This question is discussed in section 5.2. And finally, with many possible symbolic expressions that can represent

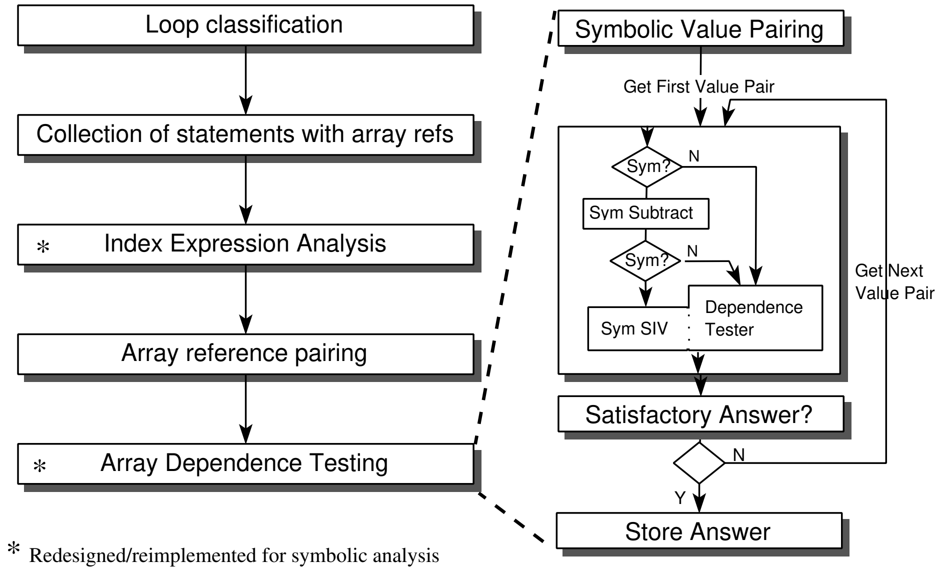


Figure 5.1: Symbolic dependence testing in the framework of array dependence analysis.

the index expression of an array reference, which possibilities should be chosen to be the result of the dependence test between the two array references. This topic is discussed in section 5.3.

5.1 Possible Value Pairing

Before starting to look at specialized dependence tests for test pairs with symbolic unknowns, it is necessary to decide how to test two array references with PSV trees as their index expressions. A PSV tree contains a hierarchical representation of the possible ways an expression can be written. At the top node is an expression that represents the whole index expression. If it has any children nodes then the index expression is alternatively represented by all of the children nodes put together. Alternatively one of these children may be similarly replaced by its children thus making a larger list of expressions that as a whole represent the index expression. Figure 5.2 highlights the step that takes this property into consideration and builds a plan for testing a pair of array references.

Let us step back a little and consider a simpler situation. Suppose that the values of the index expressions are known as integer constants (eg. `a[1]` vs. `a[5]`) or the index expressions were traced back to a single possible expression (eg. `a[1+i]` vs.

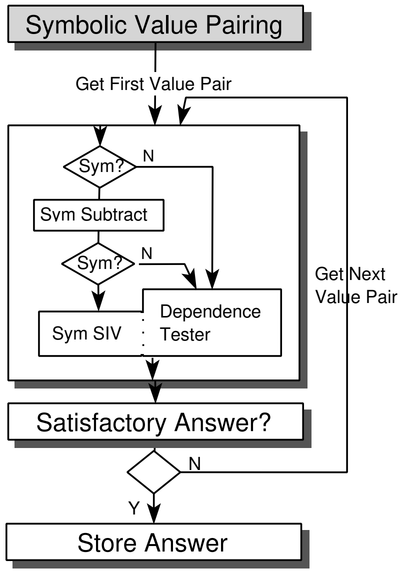


Figure 5.2: Possible Value Pairing in the context of Symbolic Dependence testing.

`a[2+2i]`). The plan for testing such an array pair is then trivial: "Send the two index expressions to the Dependence Tester with the ranges of values of loop variables." The outcome of this test will describe all the dependencies that possibly exist between the two array references.

The difficulty comes if one or both of the index expressions have multiple possible values. Consider the example in Figure 5.3. Part (a) shows the source program and part (b) shows the internal representation (PSV trees) of the index expression of the three array references.

First consider testing the array reference in statement *S61* with that of statement *S62* (`a[i+x]` vs. `a[i+1]`). Certainly it is possible to just consider the top node of the PSV tree for `i+x`. This would then require just sending the values "`i+x`" and "`i+1`" to the dependence tester. Because both of the values fully represent their corresponding index expression the result of the dependence tester can be taken as a final result. It is also possible to send the values "`i+2`" (the contents of the bottom left node of the PSV tree for array reference in statement *S61.*) and "`i+1`" to the dependence tester. This time its answer is only partial, because "`i+2`" does not fully represent the index expression in statement *S61.* The index expression could also be "`i+3`". In order to have a complete dependence test it is necessary to make two tests: "`i+2`" vs. "`i+1`" and "`i+3`" vs. "`i+1`". Both results must be kept and both must be considered when examining the dependence between the two array references.

```
      if(...)
        x = 2;
      else
        x = 3;
      for(i=0;i<100;i++)
      {
S61:    a[i + x] = ...;
S62:    a[i + 1] = ...;
S63:    a[i + x + 1] = ...;
      }
```

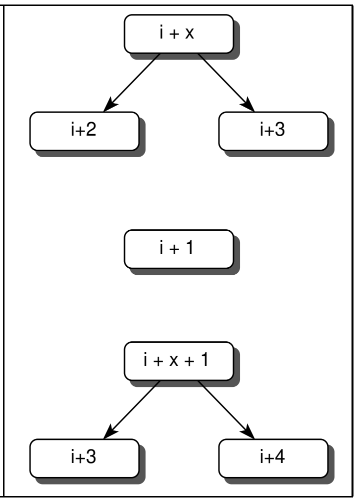


Figure 5.3: Example loop with three array references with symbolic unknowns.

Immediately the questions: "Which complete dependence test should be performed?" and "Which complete dependence test should be kept?" come to mind. This issue will be discussed in section 5.3, after discussing symbolic dependence tests.

To continue with the planning which dependence question to send to the dependence tester, one can say that the plan for testing the array reference in statements *S61* and *S62* is:

"`i+x`" vs. "`i+1`"
OR
"`i+2`" vs. "`i+1`" AND "`i+3`" vs. "`i+1`"

What would be the plan for testing the array references in statements *S61* and *S63*?

"`i+x`" vs. "`i+x+1`"
OR
"`i+x`" vs. "`i+3`" AND "`i+x`" vs. "`i+4`"
OR
"`i+2`" vs. "`i+x+1`" AND "`i+3`" vs. "`i+x+1`"
OR
"`i+2`" vs. "`i+3`" AND "`i+2`" vs. "`i+4`" AND "`i+3`" vs. "`i+3`" AND
"`i+3`" vs. "`i+4`"
OR
"`i+2`" vs. "`i+3`" AND "`i+2`" vs. "`i+4`" AND "`i+3`" vs. "`i+x+1`"
OR
"`i+2`" vs. "`i+x+1`" AND "`i+3`" vs. "`i+3`" AND "`i+3`" vs. "`i+4`"

The last two lines of the plan are quite tricky to find but they are valid. With a little bit of careful examination of the PSV trees, one can convince oneself that they are valid.

5.1.1 Building the plan

Having presented, by example, an overview of building pairings to be tested for a given pair of PSV trees, let me now present the algorithm for doing this.

To make it easier to talk about pairs let us represent each node of the PSV tree by a letter or number. Nodes of the PSV tree of the first array reference will be represented by letters, and the nodes of the other PSV tree will be represented by numbers. Figure 5.4 shows the PSV trees of array references from statements *S61* and *S63*, labeled as usual (with expressions as node contents) and labeled by letters and numbers. The PSV trees are identical in shape except that the labeling of nodes is now much more compact. With such labeling, testing "A1" means that the values "`i+x`" vs. "`i+x+1`" are sent to the dependence tester. Also a plan of "B1 C1" means that both "B1" and "C1" tests must be performed. This gives some abstraction to the problem as well as makes writing down the plan much more compact.

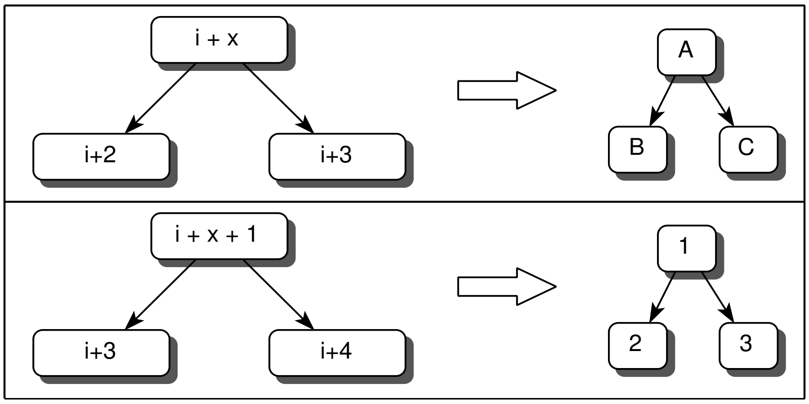


Figure 5.4: Abstracting the structure of PSV trees for the purpose of building dependence test plan.

The algorithm for generating a plan given two PSV trees is presented in Figure 5.5. The algorithm is started with the pairing "A1", which represent the pairing of the two top nodes of the two PSV trees. The function `save()` will keep a list of possible pairings. If the argument to `save()` has not been seen then `save()` will record it and return `true`; otherwise it will return `false`. The reason for returning some value is for efficiency. With this algorithm duplicate plan lines will be generated. If a plan line was already examined, then its descendants will also have been examined; therefore, the algorithm can safely stop exploring this branch.

```
fun generate_plan(plan_line)
    save(plan_line)
    foreach symbol s in plan_line
        n = get PSV node associated with symbol s
        if(has_children(n))
          new_plan_line = copy(plan_line)
          replace s in new_plan_line by "(c1 c2 ... ck)"
                  where ci is the i^th child of n
          distribute(new_plan_line)
          generate_plan(new_plan_line)
```

Figure 5.5: Algorithm to generate Possible Value Pairings

The algorithm continues, taking one of the symbols of the plan line and replaces the symbol by its children. In the case of the example, this would be replacing "A" by its children "(B C)". At this point the new plan line would be "(B C)1". The algorithm then applies simple algebraic distribution, just as if space was the algebraic "+" and adjacent symbols imply multiplication. Applying distribution to "(B C)1" will produce "B1 C1". The algorithm then continues recursively. Trying to replace "B", the algorithm fails because "B" does not have any children so it goes on to the symbol "1". This gives "B(2 3) C1" $\Rightarrow$ "B2 B3 C1". The algorithm ends up with the list of plan lines as shown below

A1
B1 C1
A2 A3
B2 B3 C1
A2 B3 C3
B2 B3 C2 C3

Even though a check has been inserted to not follow already examined paths the algorithm would still generate duplicates for the plan line "B2 B3 C2 C3". It is possible that a more complex algorithm could be designed to generate all possible plan lines without generating any duplicate entries. With such an algorithm extreme care must be taken not to miss any possible plan lines. For this reason, I have decided to stick with this simple algorithm making the `save()` function as efficient as possible. The table storing the saved plan lines, groups the plan lines by size and orders them to make lookup as fast as possible.

Another detail to this algorithm is its growth. The number of plan lines grows extremely fast when compared to the complexities of the PSV trees. For this reason, limiting factors have been added to the algorithm. The number of plan lines as well as the width of plan lines can be limited by input parameters. The only plan line that is not affected by these limiting factors is the last one. This line is composed of only leaf nodes. It can be possible that the PSV trees are very complex but their leaf nodes contain only loop variables and integer constants. It would be a shame to have overlooked such a set of tests in a search of proving dependencies between array references.

Further examining the growth aspect of this algorithm one can notice that the number of tests that can be performed in the worst case (ie. all plan lines result in unsatisfactory dependence results) is not what was promised in section 4.4.3. The number of possible expressions for both PSV trees is 2. The claim was that the number of tests that would be performed at most should be $4n^2 - 4n + 1$. In this example, $n = 2$, so that means at most 9 tests, but the number of tests in the plan is 15. What went wrong? It was not considered that the plan has plan lines that repeat tests pairs. For example, "B3" appears in 3 plan lines. It must remain in all 3 of those plan lines, but the dependence test need only be performed once with the results reused for other plan lines. Counting the number of unique tests that need to be performed for the example gives exactly 9, just as promised.

Expanding this algorithm for multi-dimensional array is trivial. For two dimensional array, all that needs to be done is change our starting input from "A1" to "A1H8" where "H" represents the top node of the PSV tree of the second dimension index expression of the first array reference, and "8" represents the top node of the PSV tree of the second dimension index expression of the second array reference.

5.2 Symbolic Dependence Testing

Having developed a plan on which value pairs need to be tested, I can now describe the process of dependence testing these value pairs in the presence of symbolic values.

Most of the process is described by the small flow chart (Figure 5.6) representing this part of the dependence tester. The most important parts are the module to perform symbolic subtraction, and several new testers to handle ZIV and SIV test pairs.

The dependence test between two test pairs is performed according to the following steps.

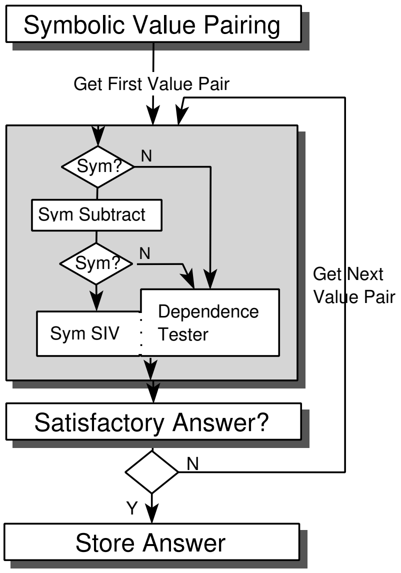


Figure 5.6: Symbolic Simplifications and ZIV and SIV tests in the context of Symbolic Dependence testing.

1. Determine if the input test pairs for each dimension of the array contain any symbolic unknowns.

   (a) If no unknown symbols are present, then send the test pairs to standard dependence tester.

   (b) If unknown symbols are present, then continue to next step.

2. Perform symbolic simplifications on additive components of index expressions to try and cancel any unknown symbols. See section 5.2.1 for a description of the implemented symbolic simplification. If successful, the test pair, now without any symbolic terms, can be sent to the standard dependence tester; otherwise, continue to next step.

3. Classify test pairs and perform symbolic dependence test if test pair requires ZIV or SIV tests. More complicated symbolic tests were not implemented and therefore a conservative assumption of possible dependence is taken. See section 5.2.2 for a description of the implemented dependence tests.

5.2.1 Symbolic Simplifications

Given two possible symbolic values from two array references that contain symbolic unknowns, one can sometimes simplify these two values so that they no longer contain symbolic unknowns and still produce the same dependence results. For example, the two symbolic values "`i+x`" and "`i+x+1`", coming from two array references located in a loop nest with `i` as one of the loop variables, can be simplified to comparing "`i`" vs. "`i+1`", as long as the symbol `x` remains constant in the entire execution of the loop nest. This can be done because no matter what the value of `x`, the dependence distance will always be -1. For the purpose of calculating dependence distance, it can be assumed that `x` has value 0, thus cancelling it out of the equations.

This simplification can be performed on any additive term as long as it remains constant throughout the entire execution of the loop nest being analyzed. This simplification is performed by following the steps:

1. Transform the two symbolic values into canonical form. This step is also required for dependence testing, and can be done at this time.

   For each loop variable in the loop nest

   (a) Factor out the loop variable out of the symbolic expression, keeping the remainder and multiplier separate.

   (b) Store the multiplier expression and the associated loop variable for lookup purposes.

   (c) For next iteration use the remainder as the symbolic expression

   The left over remainder is the additive component of the symbolic value. It is also stored as part of the canonical form.

2. Tag all symbols in the additive components of the second symbolic value that are not loop invariant. These tagged symbols will now appear as different symbols to the symbolic subtraction in the next step.

3. Subtract the additive components from each other leaving any terms with tagged symbols un-subtracted.

4. The canonical forms may now possibly be free of symbolic unknowns.

An example of the above process is shown in Figure 5.7 First the symbolic values are put into canonical representation where, factors for each loop variable are separated and stored. Next, based on the fact that `y` is not loop invariant it is tagged in

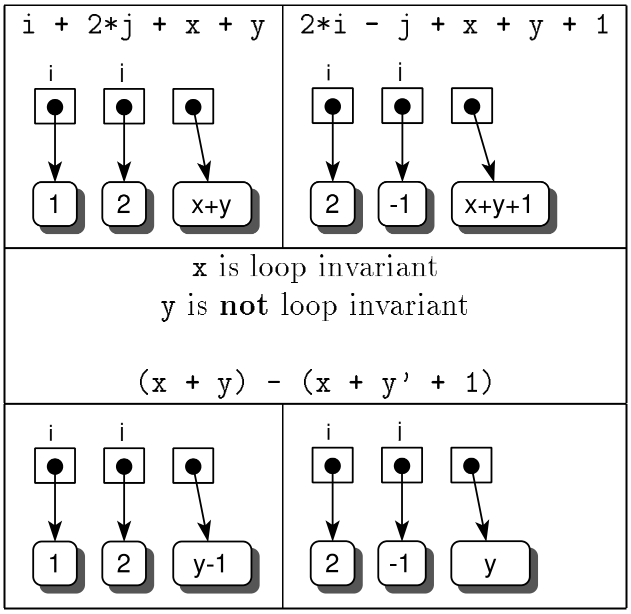


Figure 5.7: Example of symbolic subtraction simplification for the symbolic value pair "`i+2*j+x+y`" vs. "`2*i-j+x+y+1`"

one of the additive components (shown using `y'`). The additive components are then subtracted from each other cancelling the `x` symbol but leaving the `y` and `y'` in their original place. Note that the integer constant `+1` moved from one side to the other, becoming `-1`. This will make no difference in determining the dependence distances.

Can symbolic constants that are factors of loop variables be canceled? For example, is it valid to simplify "`x*i+1`" vs. "`x*i+i`" into "`1`" vs. "`i`"? The answer is no, because `i` is a loop variable and its value is different at different loop iterations which are considered by array dependence testers.

5.2.2 Symbolic ZIV and SIV array dependence tests

In many cases, symbolic subtraction of additive components is not enough to reduce index expressions containing symbolic unknowns into ones containing only loop variables and integer constants. For this reason, special symbolic versions of Zero Index Variable (ZIV) and Single Index Variable (SIV) dependence tests had to be added to the existing dependence tester.

The following were added

- Zero-index-variable (ZIV)

- Strong-SIV
- Weak-zero-SIV
- Weak-crossing-SIV
- Weak-SIV

In the case of multi-dimensional arrays the standard ZIV and SIV tests propagate constraints to next dimensions. Any contradiction in constraints leads to a conclusion of independence. Propagation of such constraints was not done in the symbolic versions of these tests.

For the purpose of discussing each of these tests, let us map the problem onto the pair

$$a_1 i + c_1 \text{ vs. } a_2 i + c_2$$

where i is the loop variable, a_x are integer constants, and c_x are symbolic expressions not involving any loop variables. More detailed description of these tests in their standard forms can be found in [GKT91, Jus95].

ZIV

The ZIV test is invoked when $a_1 = 0$ and $a_2 = 0$. The problem is therefore reduced to checking if the expressions c_1 and c_2 can possibly be equal. Because symbolic unknowns are involved, an attempt at determining the minimum and maximum values of each expression are made using the information available from generalized constant propagation. If the ranges overlap, then there exists a possibility of dependence; otherwise independence can be declared.

An example output from McCAT showing possible dependence for a ZIV pair is given in Figure 5.8. Please refer to appendix B for the description of output conventions.

Strong-SIV

The Strong-SIV test is invoked when $a_1 = a_2$. An attempt is made to try and symbolically compute $d = (c_1 - c_2)/a_1$. If successful, d is the dependence distance. If d cannot be successfully computed due to known integer indivisibility, the pair can be declared independent. If d cannot be computed due to difficulty with symbols,

```
        for ( i/*1*/ = 0 ; (i/*2*/ <= 99) ; i/*3*/ = (i/*2*/ + 1) )
        {
/*S1*/    A/*2*/[0] = 5;
/*S2*/    A/*3*/[x/*1*/] = 7;
        }

/**************************************
Summary of Array Dependence Information
**************************************
Loop variable : i(2)   Low-bound : 0 -- Upper-bound : 99


Possible Dependence Info between :S1 - S2 are :
The index PSV for LHS array reference are:
    Dimension 1:
01: 0
The PSV for RHS array reference is:
    Dimension 1:
01: + 1*x(1)

     Possible Canonical Pair : 1
Original Canonical pairs
LHS_Subscript :  0  RHS_Subscript :  + 1*x(1)
Subscript-pair type : SURE-PAIR
Dependence-pair type : FLOW-or-ANTI pair
Result : MAYBE-DEPENDENT with Distance Vector : [*]
Dimension 0 -  Dependence tester :  Symbolic ZIV test
```

Figure 5.8: Example output from McCAT analyzing a symbolic ZIV dependence.

a conservative assumption of possible dependence must be assumed. Further, the min/max range of d can be checked if it overlaps the range of the loop bounds. If it does not, that means the dependence distance is greater then the number of loop iterations thus eliminating the possibility of dependence. If it does, then the dependence distance is reported as a symbolic expression.

Example output of such a symbolic distance is shown in Figure 5.9.

Weak-Zero-SIV

This test applies when either $a_1 = 0$ or $a_2 = 0$. Similar to the Strong-SIV test, an attempt to compute $d = (c_2 - c_1)/a_1 - i_2$ (or $d = (c_1 - c_2)/a_2 - i_1$ if $a_1 = 0$) is made. Similarly independence can be declared if indivisibility is known or dependence distance can be shown out of range of loop bounds.

Example output of such a symbolic distance is shown in Figure 5.10. McCAT reports such a dependence using $i_1 = (c_2 - c_1)/a_1$ (or $i_2 = (c_1 - c_2)/a_2$ if $a_1 = 0$). This is equivalent because by definition $d = i_2 - i_1$.

```
        for ( i/*1*/ = 0 ; (i/*2*/ <= 99) ; i/*3*/ = (i/*2*/ + 1) )
        {
/*S1*/    A/*2*/[i/*2*/] = 5;
          temp_14/*2*/ = (i/*2*/ + x/*1*/);
/*S2*/    A/*3*/[temp_14/*2*/] = 7;
        }

/*************************************
Summary of Array Dependence Information
**************************************
Loop variable : i(2)   Low-bound : 0 -- Upper-bound : 99


Possible Dependence Info between :S1 - S2 are :
The index PSV for LHS array reference are:
    Dimension 1:
01: + 1*i(2)
The PSV for RHS array reference is:
    Dimension 1:
01: + 1*i(2) + 1*x(1)

     Possible Canonical Pair : 1
Original Canonical pairs
LHS_Subscript :  0 + 1*i(2)  RHS_Subscript :  + 1*x(1) + 1*i(2)
Subscript-pair type : SURE-PAIR
Dependence-pair type : FLOW Dependence
Result : SURE-DEPENDENT with Distance Vector : [ - 1*x(1)]
                        Dependence Vector : [*..*]
Dimension 0 -  Dependence tester :  Symbolic SIV_Strong test
```

Figure 5.9: Example output from McCAT analyzing a Strong-SIV dependence with symbolic distance.

Weak-Crossing-SIV

This test applies when $a_1 = -a_2$. This test is very similar to Weak-Zero-SIV except that the formula for distance is $d = -2i_1 + (c_2 - c_1)/a_1$. Example output of such a symbolic distance is shown in Figure 5.11.

Weak-SIV

This test applies when the other conditions do not hold. In other words, both a_1 and a_2 are not zero, and are not the same, and are not opposite each other. For this test, the dependence is reported using a linear equation involving both i_1 and i_2. The only chance for proving independence is if $c_1 - c_2$ can be shown to be not divisible by the greatest common divisor of a_1 and a_2. This comes from the fact that there are no solutions to the following linear Diophantine equation $ax + by + c = 0$ if c is not divisible by the greatest common divisor of a and b. Example output of a symbolic distance is shown in Figure 5.12.

```
        for ( i/*1*/ = 0 ; (i/*2*/ <= 99) ; i/*3*/ = (i/*2*/ + 1) )
        {
/*S1*/    A/*2*/[4] = 5;
          temp_14/*2*/ = (i/*2*/ + x/*1*/);
/*S2*/    A/*3*/[temp_14/*2*/] = 7;
        }

/*************************************
Summary of Array Dependence Information
**************************************
Loop variable : i(2)   Low-bound : 0 -- Upper-bound : 99


Possible Dependence Info between :S1 - S2 are :
The index PSV for LHS array reference are:
    Dimension 1:
01: + 4
The PSV for RHS array reference is:
    Dimension 1:
01: + 1*i(2) + 1*x(1)

     Possible Canonical Pair : 1
Original Canonical pairs
LHS_Subscript :  + 4  RHS_Subscript :  + 1*x(1) + 1*i(2)
Subscript-pair type : SURE-PAIR
Dependence-pair type : FLOW Dependence
Result : SURE-DEPENDENT with Distance Vector : [i(2){right}= + 4 - 1*x(1)]
                        Dependence Vector : [*..*]
Dimension 0 -  Dependence tester :  Symbolic SIV_Weak_Zero test
```

Figure 5.10: Example output from McCAT analyzing a Weak-Zero-SIV dependence with symbolic distance.

5.3 Satisfactory Answers

The last step of performing symbolic array dependence testing (Figure 5.13), is deciding when to stop examining possible series of dependence tests, for the purpose of analyzing the dependence between two array references. As discussed in section 5.1, given two PSV trees representing index expressions of two array references, such as the ones found in Figure 5.3, there are many series of tests that describe the dependence situation. For example, one could test node "A" vs. node "1" and get the conclusions, or perform two tests: node "B" vs. node "1" and node "C" vs. node "1". Both of these series of tests fully represent the dependence situation among the two array references. Which one should be chosen? Certainly it is not known which one to chose before performing the tests, but it may be decided, that after performing a few tests, there is no use performing any more tests. Either independence has been proven, or with high probability it is known that a better answer will not be generated.

```
        for ( i/*1*/ = 0 ; (i/*2*/ <= 99) ; i/*3*/ = (i/*2*/ + 1) )
        {
          temp_14/*2*/ = (i/*2*/ + 4);
/*S1*/    A/*2*/[temp_14/*2*/] = 5;
          temp_15/*2*/ = (x/*1*/ - i/*2*/);
/*S2*/    A/*3*/[temp_15/*2*/] = 7;
        }

/**************************************
Summary of Array Dependence Information
**************************************
Loop variable : i(2)   Low-bound : 0 -- Upper-bound : 99


Possible Dependence Info between :S1 - S2 are :
The index PSV for LHS array reference are:
    Dimension 1:
01: + 4 + 1*i(2)
The PSV for RHS array reference is:
    Dimension 1:
01: - 1*i(2) + 1*x(1)

     Possible Canonical Pair : 1
Original Canonical pairs
LHS_Subscript :  + 4 + 1*i(2)  RHS_Subscript :  + 1*x(1) + -1*i(2)
Subscript-pair type : SURE-PAIR
Dependence-pair type : FLOW Dependence
Result : SURE-DEPENDENT with Distance Vector : [d= - 4 + 1*x(1) + -2i(2){left}]
                        Dependence Vector : [*..*]
Dimension 0 -  Dependence tester :  Symbolic SIV_Weak_Crossing test
```

Figure 5.11: Example output from McCAT analyzing a Weak-Zero-SIV dependence with symbolic distance.

I chose to apply a heuristic on deciding when a satisfactory answer has been reached. The plan of tests, as described in section 5.1, is to run one plan line at a time, starting with the shortest lines. At each step, a heuristic evaluation is performed to see if further tests should be performed. Once a satisfactory plan line is chosen, its results are stored as possible situations. All situations must be considered when considering the dependence between the two array references.

The heuristic is based on three possible outcomes of an individual dependence test. Recall, that a dependence between two array references may require several individual dependence tests. This happens when one or both of the index expressions, expand to more then one possible value. The three possible outcomes are:

1. Independent. This is nicest result one could get. It is definitely a satisfactory result.

2. Dependent with integers or loop variables in the distance vector. This is what is generated for test pairs not requiring symbolic analysis. There is no way to

```
        for ( i/*1*/ = 0 ; (i/*2*/ <= 99) ; i/*3*/ = (i/*2*/ + 1) )
        {
          temp_14/*2*/ = (3 * i/*2*/);
/*S1*/    A/*2*/[temp_14/*2*/] = 5;
          temp_16/*2*/ = (2 * i/*2*/);
          temp_15/*2*/ = (temp_16/*2*/ + x/*1*/);
/*S2*/    A/*3*/[temp_15/*2*/] = 7;
        }

/**************************************
Summary of Array Dependence Information
**************************************
Loop variable : i(2)   Low-bound : 0 -- Upper-bound : 99


Possible Dependence Info between :S1 - S2 are :
The index PSV for LHS array reference are:
    Dimension 1:
01: + 3*i(2)
The PSV for RHS array reference is:
    Dimension 1:
01: + 2*i(2) + 1*x(1)

     Possible Canonical Pair : 1
Original Canonical pairs
LHS_Subscript :  0 + 3*i(2)  RHS_Subscript :  + 1*x(1) + 2*i(2)
Subscript-pair type : SURE-PAIR
Dependence-pair type : FLOW Dependence
Result : SURE-DEPENDENT with Distance Vector : [d= - 1*x(1)+2i(2){left}-1i(2){right}]
                        Dependence Vector : [*..*]
Dimension 0 -  Dependence tester :  Symbolic SIV_Weak test
```

Figure 5.12: Example output from McCAT analyzing a Weak-SIV dependence with symbolic distance.

improve such a result other than through loop transformations; therefore, this outcome is satisfactory.

3. Dependent with symbolic values in the dependence vector. This outcome does tell something about the possible dependence but the exact dependence distance is not known at compile time. There is a possibility that dependence tests further down in the plan, (coming from lower nodes of the PSV tree) will eliminate the symbolic unknown from the dependence vector. This outcome is therefore not completely satisfactory.

4. Conservative assumption due to non-linear index expression or presence of symbolic value not constant throughout the execution of the loop nest. Such an outcome is very bad, because absolutely no information is available about the dependence. This outcome is undesirable.

Based on these four possible types of outcomes, each line of the plan is given a

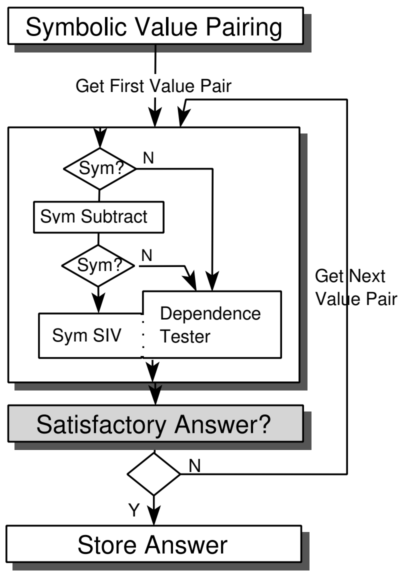


Figure 5.13: Module for deciding on a satisfactory answer in the context of symbolic dependence testing.

heuristic value. This value is based on the outcomes of each test pair in the plan line. Each test pair resulting in independent, or dependent without any symbols in the distance vector, (i.e. outcome types 1 and 2) receives the value 0. Each test pair resulting dependent with symbols in the distance vector, (i.e. outcome type 3) receives a value of 1, and each conservative assumption receives the value $\top$ (top). The values assigned to each test pair are summed ($\top$ plus anything is $\top$). Dependence testing stops if a plan line evaluates with a heuristic value of 0, and continues otherwise. The plan line with the lowest heuristic value is chosen as the result.

Different methods of selecting the most desirable plan line can be adopted, but the following should be kept in mind when choosing a heuristic.

- The four outcome types are ordered in preferred result type. Certainly one would prefer to see the result of independent then conservative dependence. Similarly a dependence with distance vector composed of only integer constants is preferred to one with symbolic unknown.
- A plan line (thus a result set) with fewer possible dependence scenarios is preferable the one with more.
- Which is preferred; a plan line with 10 outcomes of type 2, or a plan line with just one outcome, but of type 3?

Of course, this algorithm is based on a heuristic, and one cannot evaluate how well the heuristic works without experimental results. Section 7.3 examines the overall effectiveness of the PSV trees.

Chapter 6

Hierarchical Array Dependence Analysis

I have described the complete symbolic framework for array dependence testing; however, it is difficult to resist the temptation to go off topic for a while and explore other uses of the symbolic framework. Having developed a way to test for dependencies in array references containing symbolic unknowns within the context of structured loops, one can ask if dependencies can be analyzed outside of loops. Are there applications for testing dependencies outside of loops?

6.1 Introduction and Motivation

Traditional array dependence analysis is performed inside structured loops such as the one in Figure 6.1, computing Fibbonaci numbers.

```
S64: A[0] = 1;
S65: A[1] = 1;
S66: for(i=2;i<n;i++)
     {
S67:   x = A[i-1];
S68:   y = A[i-2];
S69:   A[i] = x + y;
     }
```

Figure 6.1: Example loop with initialization code.

Traditional array dependence analysis will be performed on the loop labeled *S66*. It will produce two dependencies. A flow dependence from statement *S69* to statement

S67 with a dependence distance of 1, and a flow dependence from statement *S69* to statement *S68* with dependence distance of 2. This information is very useful when generating code for the loop, but what information is present for statements *S64* and *S65*? Read/write sets will produce that statement *S64* writes `A` and statement *S65* writes `A` as well. Using just this information one would have to conclude output dependence.

The ability to schedule array references outside of the context of loops is very weak. With only read/write sets available, interference must be concluded. In the case of this example, the array reference are resolved to memory addresses at compile time and thus easily determined to be different and not interfering. However, in more complicated situation perhaps the symbolic framework can help out.

6.2 Analysis Objectives

The goal of this analysis is to be able to determine array dependencies between array references outside of admissible loops. Certainly this is an impossible task if it must be done accurately and in all situations. Because of this, the analysis will have to be very conservative when asked about unstructured (inadmissible) loops. It will be perfectly acceptable for the analysis to respond "possibly dependent" when asked about two statements representing two unstructured loops. However, it would be nice for the analysis to be able to identify that statements such *S70*, *S71* and *S74*, from Figure 6.2, are all independent.

```
S70: A[n] = ...;

S71: if(...)
S72:    A[n+1] = ...;
     else
S73:    A[n-1] = ...;

S74: for(i=0;i<n-2;i++)
S75:    A[i] = ...;
```

Figure 6.2: Example code where all statements should be determined independent.

6.2.1 Testable pairs

Notice that statements *S72*, *S73*, and *S75* were not mentioned when trying to determine interference between statements. The main application of this analysis will be to determine if order of execution between statements needs to be preserved. Because of this, only statements at the same hierarchical statement level are considered. Interchanging statements at different hierarchical levels may result in incorrect program if compensating actions are not taken. For example, if one wished to interchange statement *S70* with statement *S72* and the dependencies permitted, but because of dependencies one could not interchange *S70* with *S73*, it would be necessary to make two copies of statement *S70* and place each copy inside the two possible paths through the conditional. Similarly statement *S75* is not considered, instead its "parent" statement (*S74*) is considered.

Can statements *S72*, *S73*, and *S75* be tested with each other. Even though these statements are all in the same hierarchical level they are not part of the same immediate containing block statement. One would never test *S72* with *S73*, because they are in separate branches of their containing conditional and therefore can never both be executed in a single execution of the conditional.

In general, the array dependence test can only be performed for any two statements at the same hierarchical level and in the same statement list.

6.2.2 When to perform dependence tests?

The array dependence analysis, as implemented in McCAT, tests all possible array pairs in admissible loops and the results of the tests are stored with the loops. For hierarchical array dependence analysis testing all possible array pairs is not tractable. In many cases, there may not be any use of testing some array pairs because interference is caused by other array references or even scalar variables. It is therefore a design decision to make hierarchical array dependence analysis an analysis on demand. Only the portion of array reference summary described in the following section will be done without an explicit demand. The actual dependence testing of two array references, or more generally, two statements will be performed on demand.

6.3 Array Reference Summary

It was already suggested, in the previous section, that compound statements will have to somehow summarize array reference information. A block of assignment statements will have to somehow describe all the array references that are made by it. Similarly conditionals and loops will have to summarize all the possible array references that could be made by them if they are executed.

6.3.1 Assignment Statements

First, let us consider a single assignment statement that makes one or more array references. For each array reference a short descriptor of the array reference is built. It will contain information such as the array being referenced, PSV trees for the index expressions, and whether the array is being written or read. These descriptors can form a set that attached to each assignment statement describing its array references.

6.3.2 Statement blocks

Next, consider several assignment statements in a statement block. A simple way to summarize the array references in this block is to just union the array summary sets from all the statements in the block. This is an effective and fast method, but it may be costly on memory space.

An alternative is to group all array references by array name. Each group can be then represented by one summary descriptor stating that that whole array is being referenced. This alternative is certainly more space efficient, but not very accurate. There are intermediate alternatives, that try and describe a region of the array that is referenced when the values of index expressions are know at compile time. Much work has been done in this field and some of this work is described in section 8.3.

Because this thesis is interested in using the capabilities of the symbolic framework, it is desirable to conserve symbolic values. Also, array references with symbolic index expressions would make summary techniques very difficult, if not impossible. The array reference summaries may end up with conservative summaries stating the reference is to the whole array. With these reasons in mind, when implementing the Hierarchical Array Dependence Analysis I chose to just union the sets of statements without applying any summary techniques.

6.3.3 Conditionals

Next, let's consider the individual compound statements. Conditionals can be summarized just as blocks of statements, taking the union of the sets for each possible path. If desirable, the references in a conditional may be labeled as possible references instead of definite references such as in the assignment statement.

6.3.4 Loops

There are two types of loops that need to be considered. First let's consider loops that are part of admissible loop nests. These loops are well "behaved" with a well defined range of values for their loop variables. Suppose that the task is to summarize array references for a loop that is part of an admissible loop nest. To summarize the array references, the summary set from the body of the loop taken and to each element the piece of information that describes the range of values of the loop variable. Take for example the admissible loop nest in Figure 6.3.

```
S76: for(i=0;i<=n;i++)
S77:    for(j=0;j<=n;j++)
        {
S78:       A[i][j] = ...;
S79:       A[i][j+1] = ...;
        }
```

Figure 6.3: Example admissible loop nest to illustrate array reference summary for loops.

The summary for statement *S78* would be "Writes `A[i][j]`", and for statement *S79* it would be "Writes `A[i][j+1]`"[1]. To generate a summary for statement *S77* the summary for the loop body would be used, which is "Writes { `A[i][j]` , `A[i][j+1]` }" and to it the information that the variable `j` has values 0 to `n` would be added. The resulting summary will then be "Writes { `A[i][j]` where `j` in [0,`n`], `A[i][j+1]` where `j` in [0,`n`] }". Similarly the summary for statement *S76* would be "Writes { `A[i][j]` where `j` in [0,`n`] and `i` in [0,`n`], `A[i][j+1]` where `j` in [0,`n`] and `i` in [0,`n`] }".

Generating an accurate summary for inadmissible loops is not possible. If it was possible, then the loop would not be inadmissible. Since generating an accurate summary for inadmissible loops is not possible, a conservative summary is generated that states that the entire array is referenced. In some cases, generalized constant propagation may be used to improve this summary.

[1]Note that the index expressions would be PSV trees instead of single symbolic expression.

6.3.5 Function Calls

Generating an accurate array reference summary for function calls is difficult. The reason for this is because the index expression analysis is intra-procedural. Some symbols in index expressions will no longer be visible. A safe and conservative assumption to make is to assume that possibly the entire array is referenced. The names of these arrays are easily available from read/write sets.

6.4 Dependence Testing

After generating a summary of array references at each statement of the input program, the hierarchical array dependence analysis becomes an analysis on demand. Other modules of the compiler may ask the hierarchical array dependence analysis how two statements, from the same hierarchical level and statement list, interfere. An alternative question to ask is if two elements of summary sets interfere. It is easy to convert the question regarding two statements into a series of questions regarding summary set elements. I therefore concentrate on resolving how to answer the dependence test of two summary set elements.

A summary set element is a single array reference described by the array being referenced and PSV trees for the index expressions. Optionally, there may also be a series of definitions of loop variables. These state the range of values that will be visited by loop variables when the array references will be made. Recall that the loop variables possibly appear in PSV trees representing index expressions.

Similar to standard array dependence analysis, the first step of testing for dependencies is to check if there is a possibility that the same array will be referenced. If that is possible then testing for dependencies must go on to consider index expressions. There are three possibilities:

- At least one of the two references is a conservative assumption of referencing the whole array.
- Both references do not contain definitions of loop variables.
- At least one of the two references has definitions of loop variables.

The first possibility is easily resolved by conservatively assuming that a possible dependence exists. The other two possibilities are described below.

6.4.1 No loop variables

First let's consider testing references where there are no descriptors of loop variables. These summary elements were generated by assignment statements, statement blocks and conditionals. To test such a pair of array references all that needs to be done is to apply the zero induction variable dependence test, in either its standard or symbolic version.

For example, suppose that the two array summary elements are "`A[0]`" and "`A[1]`''. The canonical index expressions "0" and "1" are sent to the ZIV dependence test. Its result is the result of testing the two summary elements.

In the case of symbolic index expressions, the symbolic version of the ZIV dependence test is used. For elements such as "`A[n]`" vs. "`A[n+1]`", the symbolic ZIV test will easily determine independence. On the other hand, for elements such as "`A[n]`" vs. "`A[x]`", the symbolic ZIV test may have to resort to a conservative possible dependence if the range of values for `n` and `x` are not known.

6.4.2 Loop variables present

Now consider array reference summary elements where loop variable descriptors are present. For example, a summary element such as "`A[i+1]` where `i` in [0,n]" may be tested with a summary element such as "`A[0]`". Testing such a pair may seem quite difficult but it turns out to be very easy by noticing that testing this pair is equivalent to testing the admissible loop in Figure 6.4.

```
for(i=0;i<n;i++)
{
  A[i+1] = ...;
  A[0] = ...;
}
```

Figure 6.4: Admissible loop constructed to test "`A[i+1]` where `i` in [0,n]" vs. "`A[0]`".

Hence in order to test pairs with loop variables all that needs to be done is to abstractly construct a loop nest with one loop for each loop variable. The two array references are placed in the body of the loop nest. Note that the loop is not actually constructed, just the necessary data structures to send the information of such a loop to the array dependence tester are built. The dependencies reported by the array dependence tester list all dependencies between the two array reference summary elements.

6.5 Applications of Hierarchical Array Dependence Analysis

There are many uses for this analysis which make it very attractive and worth while to explore. Three uses that were immediately considered in McCAT are *parallel statement detection*, *instruction scheduling* and *scalar replacement*.

6.5.1 Parallel Statement Detection

For multi-processor architectures that can explicitly distribute execution of independent instructions among all the available processors, it is desirable to identify all statements that can be executed in parallel. Using read/write sets one can determine this fairly easily. To test if statement `S` and statement `T` can be executed in parallel it is necessary to check *Bernstein Conditions.* [Ber66]

$$
\begin{aligned}
\text{Read}(\mathtt{S}) \cap \text{Write}(\mathtt{T}) &= \emptyset \\
\text{Write}(\mathtt{S}) \cap \text{Read}(\mathtt{T}) &= \emptyset \\
\text{Write}(\mathtt{S}) \cap \text{Write}(\mathtt{T}) &= \emptyset
\end{aligned}
$$

If all the conditions are satisfied then the two statements are *independent* meaning that they can be executed in parallel.

Using read/write sets to check Bernstein conditions will work, but for statements with array variables in read/write sets, unnecessary conservative assumption of the whole array being referenced is taken. Using the hierarchical array dependence analysis for checking Bernstein conditions can be more accurate. For example consider two assignment statements such as in Figure 6.5. The write sets for statements *S80* and

```
S80: A[n] = 0;
S81: A[n−1] = 0;
```

Figure 6.5: Checking Bernstein condition for two assignment statements.

S81 are {`A`} and {`A`} respectively. Checking the last Bernstein condition proves the two statements interfering. Instead if the hierarchical array dependence analysis to test the dependencies due to arrays between statements is used, the two statements are proven not to interfere.

Here at McGill, McCAT is being used as a compiler for a multi-processor architecture called EARTH [HMT+95, HMT+96, HTZ+96]. When compiling for EARTH, detection of parallel statements is important for effective code generation. Checking Bernstein conditions was experimentally changed to use the hierarchical array dependence analysis. The result of testing all possible statement pairs showed that most interferences are generated by scalar variables.

6.5.2 Instruction Scheduling

Considering that source level statements mostly interfere due to scalar variables motivates an investigation into finger grain parallelism. Even though the statements in Figure 6.6 interfere due to a scalar variable `x`, hierarchical array dependence analysis can determine that the loads and store to the array do not interfere. This permits the scheduler to schedule the loads earlier avoiding delays generated by memory accesses.

A[k]=x + y;
x = A[k+1] + A[k+2];
	LD r4,A[k+1]
	LD r5,A[k+2]
	ADD r1,r2,r3 // r1 = x + y
	ST A[k],r1
	ADD r2,r4,r6 // r2 = A[k+1] + A[k+2]

Figure 6.6: Example of using hierarchical array dependence information to better schedule two statements interfering at source statement level.

A key component of instruction scheduling is a Data Dependency Graph (DDG) which exposes dependencies between low level instructions. Because array references are translated to load and store instructions at the low level assembly code, these instructions may have high priority to be moved up in the schedule. The re-ordering of these instructions, to reduce stalls, may become very important when load instructions come after store instructions. In such cases, if it can be proven that the store instruction and load instruction reference different memory cells (i.e. different array elements) the load instruction will not have to wait for the store to complete before it can be issued.

When would such cases occur and when would scheduling make such an impact to justify performing an analysis like hierarchical array dependence analysis? The answer to this is loops. For structured and admissible loops this information is already available, but for other loops there is no information about array references. For example, scheduling the body of a `while` loop nest could be hindered because no information is available for array references. With the availability of the hierarchical array dependence analysis array dependencies between assignment statements may be known, and should these assignment statements be part of an inadmissible loop nest the impact can be significant.

6.5.3 Scalar Replacement

The previous two uses of hierarchical array dependence analysis made use of information stating that some pair of statements or array references are independent. One can also take advantage of the result of definite dependence. The optimizing transformation called *Scalar Replacement*[CCK90] can take advantage of some definite dependencies.

Consider the proposed transformations presented in Figure 6.7. Notice the presence of the SSA number, represented symbolically as i, at the array variable `A`. In order to permit this optimizing transformation to be valid, the array element must not be modified. An easy, but perhaps an over-conservative method is to ensure that there was no modification to any element of the array. Extended SSA numbering gives this piece of information via the SSA number of the array.

$A_i[k]$ = expr;
 	
...
x = $A_i[k]$;
...
y = $A_i[k]$;
	=>	temp = expr;
		$A_i[k]$ = temp;
		...
		x = temp;
		...
		y = temp;

x = $A_i[k]$;
 	
...
y = $A_i[k]$;
...
z = $A_i[k]$;
	=>	temp = $A_i[k]$;
		x = temp;
		...
		y = temp;
		...
		z = temp;

Figure 6.7: Two versions of scalar replacement optimizing transformation.

The implementation of this optimizing transformation has also been left for future work.

6.6 Conclusions

The hierarchical array dependence analysis is a very simple extension to any array dependence analysis that can provide benefits to several other compiler modules. Even with minimal symbolic capabilities the hierarchical array dependence analysis can help to produce better code.

Considering that most optimizing/parallelizing compilers are already equipped with array dependence analyses and hopefully soon with symbolic frameworks, the hierarchical array dependence analysis can be implemented quickly using existing components of the compiler. Making an analysis built out of existing components is an opportunity that is hard to pass up, and should be made available in all compilers with the required components already implemented.

Chapter 7

Observations and Experimental Results

After any new propositions or presentation of a new compiling technique, it is important to examine the applicability and effectiveness of the technique. This thesis presented the following techniques and extensions:

- extended SSA numbering;
- extend the number of admissible loops, by allowing function calls inside the loop body as long as the function does not make any array references or modify loop variables;
- induction variables incremented by a symbolic constant;
- possible symbolic value (PSV) trees, an internal representation to represent a single symbolic expression along with all of its possible symbolic values;
- symbolic simplifications before array dependence testing; and
- symbolic versions of ZIV and SIV dependence tests.

To show the applicability of these ideas, the McADT statistical profiler [Jus95] was used to collect profile information on several benchmarks. The McADT statistical profiler had to be modified to take into account the symbolic framework.

It is impossible to effectively analyze the performance of the entire symbolic framework for array dependence analysis. Instead, I analyze the effects of each component separately. For each component, an examination is made to show how it performed on benchmarks, a description of the parameters used to show its behavior is provided, and some observations from the results are drawn. The only exception is extended SSA numbering. Extended SSA numbering is used extensively by each component and its usefulness is hard to quantify.

Array dependence analysis is an analysis that is applied for loop nests manipulating arrays. Such loops tend to exist mostly in the core of the application. Every application must devote at least minimal effort to provide some user interface, and the user interface tends to be a large part of applications when looking at the program code statically. It is therefore difficult to find applications with large number of loop nests to make results look impressive. What is typically done, is that loop nests are extracted from many programs and put together in a test suite benchmark. Also, developers make up loop nests to test compilers for completeness. Such benchmarks are referred to as synthetic. Several programs in my benchmark list are synthetic. I always split results into two parts, with top part of the table composed of real applications, and the lower part with synthetic benchmarks.

Table 7.1 is a list of benchmarks used in this chapter. Each entry in the table describes the benchmarks using its name, a short description, and its size measured by the number of SIMPLE statements.

7.1 Admissible Loop Nests

Part of any work done on existing modules, is adding improvements to these existing modules. I have worked on the array dependence tester first written for McCAT by Justiani [Jus95]. Its organization was described in section 2.2. As mentioned in section 2.2, one of the additions was to check function calls when detecting admissible loops. Function calls that do not modify any loop variables or loop bounds, as well as do not contain any array references, will not have an effect on array dependence analysis, and therefore can be accepted for dependence testing.

In addition to accepting function calls, "good" loop detection was changed so that detection continues to inner loops even after outer loop is considered "bad". This was not very crucial in the original array dependence framework, but it becomes important once we add symbolic capabilities. Consider the short example code in Figure 7.1. In this example the statement labeled *S82* calls a function that modifies the outer loop's loop variable (`j`). This causes the outer loop to be rejected by "good" loop detection. The inner loop was skipped and the algorithm continued to analyze code after both loops. This was not important because the array references of the inner

Benchmark Name	Description	# SIMPLE Statements
`alvinn`	SPEC'92 benchmark. Program that trains a neural network called ALVINN using back propagation	275
`compress`	SPEC'92 benchmark. UNIX compress utility	1318
`ear`	SPEC'92 benchmark. Program to implement a model of acoustic propagation and detection in the human cochlea.	2342
`tomcatv`	SPEC'92 benchmark. Mesh generation with Thompsons solver. (translated to C)	
`cluster`	Graph clustering algorithm	738
`complex`	Numerical Recipes in C (complex numbers)	377
`asuite`	Test suite for vectorizing compiler. C version based on version from Argonne.	457
`csuite`	C test suite for vectorizing C compilers.	925
`clinpack`	Test suite for vectorizing compiler. C version.	1020
`nrcode1`	Application suite of routines for vectorizing C compilers.	491
`nrcode2`		253
`nrcode3`		154
`nrcode4`		125
`heap`	Heap sort algorithm test.	147
`sort`	Three sorting algorithms test. Bubble, Quick, Quick 3-median.	169

Table 7.1: Description of benchmarks used.

loop will probably contain `j` in their index expressions, such as the array references in statements *S83* and *S84*. Because `j` cannot be resolved into an integer, and `j` is not a loop variable of any "good" loop, such array pairs could not be analyzed.

```
       for(j=0;j<100;j++)
       {
S82:     f(&j); /* Bad function call */
         ...
         for(i=0;i<100;i++)
         {
S83:       A[j][i] = ...
S84:       A[j+1][i] = ...
         }
       }
```

Figure 7.1: Situation where "good" loop detection needs to continue towards inner loops.

With the presence of symbolic capabilities, one can immediately notice that `j` remains constant throughout the inner loop. It is therefore a symbolic constant and easily resolved by the symbolic framework, identifying the inner loop as independent.

Table 7.2 shows the effects of making these changes to the criteria for admissible loops. It shows the number of admissible loops and which changed identified them to become admissible. The meaning of the columns of this table are described below.

Benchmark	Admis.	With Fn	% With Fn	Inner	% Inner	Inadmis.	Bad Fn
`alvinn`	8	0	0%	1	13%	17	10
`compress`	2	0	0%	0	0%	2	0
`ear`	24	7	29%	7	29%	20	4
`tomcatv`	12	0	0%	0	0%	0	0
`cluster`	21	6	29%	5	24%	14	4
`complex`	5	5	100%	2	40%	6	2
`asuite`	172	98	58%	3	2%	13	3
`csuite`	62	36	58%	2	3%	2	0
`clinpack`	19	3	16%	0	0%	10	10
`nrcode1`	29	13	49%	0	0%	6	1
`nrcode2`	10	1	10%	1	10%	2	0
`nrcode3`	13	5	38%	1	8%	0	0
`nrcode4`	10	4	40%	1	10%	2	0
`heap`	1	1	100%	0	0%	0	0
`sort`	4	1	25%	0	0%	1	1
Total	392	180	46%	23	8%	95	35

Table 7.2: Effects of changes to admissible loopnest detection.

Admis: Number of admissible loop nests detected.

With Fn: Out of all the admissible loop nests, this column shows the number of these loop nests that contain a function call. Normally, all of these loop nests would not be admissible.

% With Fn: The percentage of admissible loop nests with function calls.

Inner: This column refers to the situation presented in Figure 7.1. The count presented in this column is the number of admissible loop nests that were detected inside inadmissible loops. Unfortunately, this count does not include loops detected inside other loop constructs such as `while` or `do-while` loops.

% Inner: The percentage of admissible loop nests that appear inside inadmissible loops.

Inadmis: This column shows the number of inadmissible loops found in the benchmark.

Bad Fn: This column shows the number of loops that were not admissible due to a function call modifying a loop variable, loop bound or containing array references.

7.1.1 Observations

Looking at these results it is easy to conclude that spending the effort on implementing the changes was worth while. Take for example the test suites `asuite` and `csuite`. The number of admissible loops has more then doubled just by checking function calls. In real applications, the effect is not as visible because of low number of admissible loops, but still in the benchmark `ear` the number of admissible loops has more than doubled.

Increasing the number of loops that can be tested is very important. For all the loops that cannot be tested, we must assume that a dependence exists and evaluate them one iteration at a time. For this reason, I have spent some time showing what can be done to improve the situation.

7.2 Induction Variables

There weren't many changes to induction variable analysis that I implemented. Induction variable analysis remained very similar to what was found in [Jus95]. The only change is the added ability to accept increments of induction variable by a symbolic value instead of only integer constants. For example consider the loop in Figure 7.2 extracted from the `clinpack` benchmark.

```
       ix = 1;
       iy = 1;
       if(incx < 0) ix = (-n+1)*incx + 1;
       if(incy < 0) iy = (-n+1)*incy + 1;
       for (i = 0;i < n; i++)
       {
         dy[iy] = dy[iy] + da*dx[ix];
S85:     ix = ix + incx;
S86:     iy = iy + incy;
       }
```

Figure 7.2: Example loop from `clinpack` benchmark.

There are two induction variables, `ix` and `iy`. They are incremented regularly at statements *S85* and *S86*. Before the addition of symbolic framework, these induction variables would not be resolved. This is because they are incremented by an unknown amount, assumed to be possibly varying. With the symbolic framework, it is known that these induction variables are incremented by a loop invariant value. It happens to be a symbol, but the symbolic framework is not afraid of that.

This all sounds great. Unfortunately there is some difficulty when considering the equivalent expression for these induction variables. The equivalent expression for `ix` will be very similar to: `incx*i +` <start-value>. A similar expression is determined for `iy`. Is this bad? The answer depends on the sophistication of the symbolic dependence analysis. Section 5.2.1 discussed if it is possible to symbolically simplify "`x*i+1`" vs. "`x*i+i`" into "`1`" vs. "`i`". The answer was no. It would therefore be necessary for the individual dependence tests to handle such situations. This may cause difficulties, because dependence tests rely on knowing the relationships between these multipliers of loop variables (eg. GCD greatest common divisor). At the moment, McCAT cannot handle such dependence tests and chooses a conservative solution of possibly dependent.

Is there a symbolic induction variable for which McCAT can perform dependence analysis? Yes. Consider an induction variable that is incremented by a constant integer amount known at compile time, but its initial value is unknown. This may produce a formula such as "`2*i + x`". The McCAT framework for symbolic dependence analysis can easily work with such an index expression.

The symbolic framework attempts to compute induction variables with or without symbols. Table 7.3 shows the number of times induction variable analysis successfully computed a formula for the induction variable as well as the times it failed. In Table 7.3 the columns are as follows:

Total IV's: The total number of induction variables that were detected. This also includes induction variables whose formula could not be derived.

Resolved: The number of induction variables for which an equivalent formula was computed.

Symbolic: Out of the induction variables for which a formula was computed, this column shows the number of induction variables that required the symbolic framework.

Benchmark	Total IV's	Resolved	Unresolved	Symbolic
`alvinn`	0	0	0	0
`compress`	0	0	0	0
`ear`	0	0	0	0
`tomcatv`	2	2	0	0
`cluster`	2	0	2	0
`complex`	0	0	0	0
`asuite`	8	2	6	0
`csuite`	0	0	0	0
`clinpack`	5	5	0	5
`nrcode1`	0	0	0	0
`nrcode2`	0	0	0	0
`nrcode3`	0	0	0	0
`nrcode4`	0	0	0	0
`heap`	0	0	0	0
`sort`	2	0	2	0
Total	19	9	10	5

Table 7.3: Summary of detected induction variables.

7.2.1 Observations

It is commonly known that induction variable analysis is very important in array dependence analysis. Failure to recognize and resolve induction variables is just as costly as failure to analyze the header of a loop making it inadmissible for dependence tests. It is therefore important to have induction variable analysis.

As developers, we often tend to do some of our own optimizations while coding. For example, I often use a temporary scalar variable to propagate a value of an expression involving indirection (common subexpression elimination). It is similar with induction variables. If it is easy for the compiler to come up with a formula for an induction variable, a good programmer will also notice the formula and use it directly. Because of this, few induction variables have been detected in structured `for` loops. Five of the six unresolved IV's were due to conditional increment and one because it was a geometric IV.

Although some of the formulas generated for induction variables with symbols cannot be used by the current implementation of the symbolic dependence analysis, it was worth while to implement the resolution of these induction variables for future symbolic dependence tests. The only difference is being able to test for loop-invariant symbols instead of just integer constants. Testing for loop-invariant symbols is required by the symbolic framework anyway.

7.3 Possible Symbolic Value Trees

Chapter 4 has spent a lot of time describing PSV trees, and thus committed some computation to build PSV trees. The question "Are PSV trees needed?" can be asked. Is it possible to not expand symbols into conditionals? For example, suppose we have a situation such as the one found in Figure 7.3 (this example was also given in Figure 5.3).

```
        if(...)
          x = 2;
        else
          x = 3;
        for(i=0;i<100;i++)
        {
S61:      a[i + x] = ...;
S62:      a[i + 1] = ...;
S63:      a[i + x + 1] = ...;
        }
```

Figure 7.3: Example loop that requires PSV trees to perform most accurate array dependence analysis.

There are two possible alternatives:

- Do not expand symbols beyond conditionals. This results is treating the symbol to have unknown value. This approach was taken by Haghighat and Polychronopoulos [HP96]. The three index expressions for statements *S61*, *S62*, and *S63* would be "`i + x`", "`i + 1`", and "`i + x + 1`" respectively. Dependence analysis would have no trouble analyzing statement *S61* vs. statement *S63*, but conservative assumptions may have to be made when testing *S61* vs. *S62* and *S62* vs. *S63*, because the relationship between `x` and `1` may not be known.

- An alternative is to expand symbols fully. This approach was taken by McCAT before the introduction of the symbolic framework [Jus95]. The result of taking this approach is that index expressions for the three statements would be:

  S61: "`i + 2`" or "`x + 3`"
  S62: "`i + 1`"
  S63: "`i + 3`" or "`x + 4`"

  This alternative sometimes leads to conservative results. In the example, *S61* vs. *S63* would be declared to possibly have a dependence of distance zero ("`x +3`" vs. "`x + 3`"), which is not true. In the worst case, if the terminal expression has symbols instead of integer constants, as in the example, there would be several conservative results because relationships between different symbols may be unknown.

Having seen the alternatives and their problems solved by the PSV trees (section 4.4), one should ask how often is it necessary to build PSV trees. Table 7.4 shows statistics about PSV trees that were built in admissible loop nests, and Table 7.5 shows statistics about PSV trees that were built for index expressions outside of admissible loops. Note that PSV trees built outside of admissible loops are used by the hierarchical array dependence tester.

Benchmark	PSV's built	PSV's in loops	Single node	Multi-node	Avg. size multi-node
`alvinn`	51	44	44	0	
`compress`	66	3	3	0	
`ear`	240	109	106	3	6.3
`tomcatv`	99	99	83	16	3.0
`cluster`	282	160	160	0	
`complex`	14	5	5	0	
`asuite`	634	564	564	0	
`csuite`	159	131	131	0	
`clinpack`	313	105	100	5	3.0
`nrcode1`	80	71	71	0	
`nrcode2`	53	48	45	3	3.0
`nrcode3`	31	30	30	0	
`nrcode4`	17	13	13	0	
`heap`	29	1	1	0	
`sort`	43	17	17	0	
Total	2111	1400	1373	27	

Table 7.4: Sizes of PSV trees in admissible loops.

The following describes the meaning of each column.

PSV's built: The total number of PSV trees built in the benchmark.

PSV's in loops: The total number of PSV trees built inside admissible loops.

PSV's outside: The total number of PSV trees built outside of admissible loops.

Single node: Out of the PSV trees built inside (outside of) admissible loops, this column shows the number of PSV trees that consisted of only one node.

Benchmark	PSV's built	PSV's outside	Single node	Multi-node	Avg. size multi-node
`alvinn`	51	7	7	0	
`compress`	66	63	59	4	3.75
`ear`	240	131	130	1	3.0
`tomcatv`	99	0	0	0	
`cluster`	282	122	122	0	
`complex`	14	9	8	1	3.0
`asuite`	634	70	70	0	
`csuite`	159	28	28	0	
`clinpack`	313	208	208	0	
`nrcode1`	80	9	4	5	3.0
`nrcode2`	53	5	3	2	3.0
`nrcode3`	31	1	1	0	
`nrcode4`	17	4	4	0	
`heap`	29	28	18	10	3.6
`sort`	43	26	17	9	4.3
Total	2111	711	679	32	

Table 7.5: Sizes of PSV trees in hierarchical array dependence analysis.

Multi-node: Out of the PSV trees built inside (outside of) admissible loops, this column shows the number of PSV trees that consisted of more then one node.

Avg. size multi-node: The average size of multi-node PSV trees that were constructed inside (outside of) admissible loops.

Table 7.4 and 7.5 show information about the building of PSV trees. They do not discuss the use of the PSV trees in dependence testing. Table 7.6 and 7.7 show when PSV trees with multiple nodes were used, an when they were a benefit to array dependence analysis.

The following describes the meaning of each column.

Array pairs: The number array pairs that contained multi-node PSV trees. Note that the hierarchical array dependence analysis used both the PSV trees built outside as well as inside of loops to collect array pairs.

Used only top node: The number of array pairs that only used the top node of its multi-node PSV trees to determine the dependencies.

Used mult. nodes: The number of array pairs that used the additional information provided by PSV trees to try and determine the array dependencies.

Benchmark	Array pairs	Used only top node	Used mult. nodes	Better result
`alvinn`	0	0	0	0
`compress`	0	0	0	0
`ear`	0	0	0	0
`tomcatv`	6	6	0	0
`cluster`	0	0	0	0
`complex`	0	0	0	0
`asuite`	0	0	0	0
`csuite`	0	0	0	0
`clinpack`	2	0	2	0
`nrcode1`	0	0	0	0
`nrcode2`	1	1	0	0
`nrcode3`	0	0	0	0
`nrcode4`	0	0	0	0
`heap`	0	0	0	0
`sort`	0	0	0	0
Total	9	7	2	0

Table 7.6: Occurrence of array pairs in admissible loops that contained/used multi-node PSV trees.

Benchmark	Array pairs	Used only top node	Used mult. nodes	Better result
`alvinn`	0	0	0	0
`compress`	6	0	6	0
`ear`	5	0	5	2
`tomcatv`	135	23	112	32
`cluster`	0	0	0	0
`complex`	4	0	4	0
`asuite`	0	0	0	0
`csuite`	0	0	0	0
`clinpack`	14	1	13	0
`nrcode1`	1	1	0	0
`nrcode2`	31	6	25	1
`nrcode3`	0	0	0	0
`nrcode4`	0	0	0	0
`heap`	184	8	176	16
`sort`	134	3	131	24
Total	514	42	472	75

Table 7.7: Occurrence of array pairs in hierarchical array dependence analysis that contained/used multi-node PSV trees.

Better result: The number of array pairs that got better result because they used the additional information provided by PSV trees.

7.3.1 Observations

Seeing the statistics, it is easy to conclude that inside admissible loops, most cases are solved by just considering the top node of the PSV tree. Therefore, if one is considering implementing such a symbolic framework, they can consider simplifying their work by building symbolic expressions without considering conditionals. However, it is important not to forget that conditional expressions must still be considered in order to resolve induction variables.

For hierarchical array dependence analysis, PSV trees are more important. The majority of better results with multi-node PSV trees are results that require fewer symbols because some symbol is resolved to be one of several constant integer values. Also, some of the better results give two exact dependencies instead of one symbolic dependence.

On the other hand, let's consider the cost of PSV trees when only the top node is created and used. The cost ends up being just a single level of indirection introduced by a tree node holding a symbolic expression. The logic introduced to handle multiple node PSV trees would never be used. The run-time cost of building a multi-node PSV turns out to negligible as compared to performing the dependence tests. With this in mind, the only significant cost of implementing PSV trees vs. not implementing them is the effort required by the developer. Thus, if maximum accuracy is required, PSV trees are worthwhile.

7.4 Symbolic Simplifications

Getting towards the dependence testing part of this framework, let us first take a look at symbolic simplification. The only simplification that was implemented was symbolic subtraction (section 5.2.1). It is one of the simplest parts of the symbolic dependence analysis. It also produces the most accurate dependence results (dependence distance is an integer constant). Depending on the results, it may be advantageous to just implement symbolic subtraction and not worry about complex symbolic dependence tests. Table 7.8 summarizes the effectiveness of symbolic subtraction on array dependence analysis.

The following is a description of the columns of Table 7.8.

Benchmark	Admis. loops	Array pairs	Subtract arr. pairs	Loops with symbols	Loops with subtract	Loops resolved by subtract
`alvinn`	8	8	1	1	1	1
`compress`	2	1	0	1	0	0
`ear`	24	68	10	6	4	2
`tomcatv`	12	37	6	2	2	2
`cluster`	21	97	17	5	5	1
`complex`	5	1	1	1	1	1
`asuite`	172	220	13	7	5	4
`csuite`	62	8	1	1	1	1
`clinpack`	19	15	0	0	0	0
`nrcode4`	29	26	0	0	0	0
`nrcode2`	10	7	1	1	1	1
`nrcode3`	13	9	5	1	1	1
`nrcode4`	10	1	0	0	0	0
`heap`	1	0	0	0	0	0
`sort`	4	23	4	2	2	2
Total	392	521	59	28	23	16

Table 7.8: Effectiveness of symbolic subtract simplification.

Admis. loops: The number of admissible loops.

Array pairs: The number of array references that were paired up for dependence analysis.

Subtract arr. pairs: Out of the array references that were paired up for dependence analysis, this column shows the number of array pairs that were successfully simplified to integer canonical form. These array pairs could then be analyzed by the standard array dependence tests.

Loops with subtract: This column shows the number of admissible loops where symbolic subtraction was successful in resolving at least one array pair.

Loops resolved by subtract: This column shows the number of admissible loops that were completely resolved by symbolic subtraction. In other words, these loops contained symbolic index expressions and possibly some integer canonical index expressions. All symbolic index expressions in these loops were resolved into integer canonical forms such that the entire admissible loop could be processed by the standard integer dependence tests.

7.4.1 Observations

It is easy to conclude that symbolic subtraction is effective; however, there are cases where the loops are not completely resolved by symbolic subtraction. For each test pair such as "`i + x`" vs. "`i + x + 1`" there tends to also be another index expression not involving the symbolic constant, in this case `x`. This leads to two other array reference pairs that will require symbolic dependence tests.

For someone designing a parallelizing compiler with an array dependence analysis, a natural question is: "What situations lead to array pairs which are resolved by symbolic subtraction?" Here are my observations about this question:

- For multi-dimensional arrays, there are cases where the outer loop is not admissible (either a bad header, or the loop construct is `while` or `do-while`). The inner loop is still analyzed, but the loop variable of the outer loop appears in an index expression of one array dimension. This index expression consists of only the outer loop variable plus an integer constant. Because the outer loop variable is considered as a symbolic constant the dependence test for that dimension, reduces to an integer ZIV test. The benchmark `asuite` is a prime example of this. One its loops is given in Figure 7.4(a). The `for` loop, with loop variable `j`, is surrounded by a loop structure that will not necessarily be recognized as a regular loop with loop variable `i`[1]. In that case the variable `i` will become constant for the duration of the `for` loop. The dependence tests will then use symbolic subtraction that will completely resolve the loop from symbols.

- Parameter variables, such as data lengths, that configure the behavior of the program often appear as additive constants in array index expressions. Such situations tend to accompany other array pairs that require symbolic tests. An example benchmarks is `ear`, which contains configuration variables such as `CorrelationFFTLength` and `CorrelationDataLength`. One such loop using `CorrelationFFTLength`, a global variable, acting as a size parameter is given in Figure 7.4(b). In this loop, consider the assignments to arrays `Real[j]` and `Im[j]` at statements *S91* and *S92*. The index expression `j` expands to `CorrelationFFTLength - i`. Matching these statements with references `Real[j]` and `Im[j]` in statements *S87*, *S88*, *S89*, *S90*, results in simplified symbols leaving just loop variables. Note that these references are also accompanied by references to `Real[i]` and `Im[i]` which will require further symbolic analysis.

[1]Some compilers manage to detect such loops are "good", but at this stage McCAT does not.

```
      i=1;
l1: if(i > n ) goto l2;
      for (j=2; j<n; j++)
      {
          aa[i][j] = aa[i][j-1]*bb[i][j-1];
          bb[i][j] = aa[i][j-1]*bb[i][j-2];
          cc[i][j] = cc[i][j-1]*cc[i][j-2];
      }
      i++;
      goto l1;
l2:
```

(a)

```
      /* CorrelationFFTLength is a global parameter */
     for (i=0;i<=CorrelationFFTLength/2;i++)
     {
      j = CorrelationFFTLength - i;
S87:  FirstReal =  (Real[i] + Real[j]) /2.0;
S88:  FirstIm =     (Im[i]   - Im[j])   /2.0;
S89:  SecondReal = (Im[i]   + Im[j])    /2.0;
S90:  SecondIm =    (Real[i] - Real[j]) /2.0;

      FirstMag = FirstReal*FirstReal + FirstIm*FirstIm;
      SecondMag = SecondReal*SecondReal + SecondIm*SecondIm;

      Real[i] = FirstMag;
S91:  Real[j] = FirstMag;
      Im[i] = SecondMag;
S92:  Im[j] = SecondMag;
     }
```

(b)

Figure 7.4: Example loops where symbolic subtraction is common.

7.5 Symbolic ZIV and SIV Dependence Tests

The last component of the symbolic array dependence framework that is left to analyze is the individual symbolic ZIV and SIV dependence tests. Table 7.9 shows the results of the ZIV and SIV tests. For each test, it shows the number of independent and number of dependent answers provided due to this test.

The following describes the meaning of each column.

ZIV: the number of ZIV dependence tests that resulted in either independence or possible dependence.

Benchmark	ZIV		SIV-strong		SIV-wk-zero		SIV-wk-cr		SIV-weak	
	dep.	ind.	dep.	ind.	dep.	ind.	dep.	ind.	dep.	ind.
`alvinn`	0	0	0	0	0	0	0	0	0	0
`compress`	0	0	1	0	0	0	0	0	0	0
`ear`	0	0	0	1	0	0	17	0	0	0
`tomcatv`	0	0	0	0	0	0	0	0	0	0
`cluster`	21	0	21	0	86	0	0	0	0	0
`complex`	0	0	0	0	0	0	0	0	0	0
`asuite`	0	0	1	0	2	0	1	0	0	0
`csuite`	0	0	0	0	0	0	0	0	0	0
`clinpack`	0	0	0	0	0	0	0	0	0	0
`nrcode1`	0	0	0	0	0	0	0	0	0	0
`nrcode2`	0	0	0	0	0	0	0	0	0	0
`nrcode3`	0	0	0	0	0	0	0	0	0	0
`nrcode4`	0	0	0	0	0	0	0	0	0	0
`heap`	0	0	0	0	0	0	0	0	0	0
`sort`	0	0	0	0	0	0	0	0	0	0
Total	21	0	23	1	88	0	18	0	0	0

Table 7.9: Effectiveness of symbolic ZIV and SIV dependence tests.

SIV-strong: the number of SIV-strong dependence tests that resulted in either independence or possible dependence.

SIV-wk-zero: the number of SIV-weak-zero dependence tests that resulted in either independence or possible dependence.

SIV-wk-cr: the number of SIV-weak-crossing dependence tests that resulted in either independence or possible dependence.

SIV-weak: the number of SIV-weak dependence tests that resulted in either independence or possible dependence.

7.5.1 Observations

It is clear that symbolic dependencies will result mostly in being dependent. This is because symbolic subtraction consumes many of these independent results. The few independent results were generated mostly because the dependence distance is symbolically compared with the symbolic representation of the loop upper bound. When such a check is not performed symbolically, but instead the comparison is done using integer value ranges provided by generalized constant propagation, these independent results are not found.

A result of "dependent" is not all bad. Even if the result is dependent with a symbolic dependence distance of say `x`, we could split the loop into two, where the inner loop executes `x` iterations and the outer loop repeats these until the total number of iterations is achieved. This may require a trailing loop. The result is that the inner loop may now be executed in parallel. Such transformation is called strip-mining [Wol96]. Figure 7.5 shows this transformation for a sample loop. Certainly considerations must be taken in cases where the symbol `x` is zero or negative and possibly a conditional may surround this transformation.

```
for(i=0;i<n;i++)
{
   a[i + x] = ...;
   a[i] = ...;
}
```

(a)

```
for(i=0;i<n/x;i++)
{
   for(j=i*x;j<(i+1)*x;j++)
   {
      a[j + x] = ...;
      a[j] = ...;
   }
}
/* trailing loop */
for(j=i*x;j<n;j++)
{
   a[j + x] = ...;
   a[j] = ...;
}
```

(b)

Figure 7.5: Example of strip-mining when dependence information gives symbolic dependence distance. (a) original loop, (b) transformed loop.

Another method for overcomming symbolic dependence distance is by generating different versions of the loop. At runtime the correct version of the optimized/transformed loop is selected when the value of the symbol is known. Byler et al. discusses this possibility in their paper [BDH+87].

7.6 Overall Results

In this section I present the overall effectiveness of the symbolic framework by examining how many more array reference pairs can be tested. Also, let's consider the loops as a whole. Some loops may have had array reference pairs with symbolic index expressions with some integer canonical index expressions. Although some dependence test could be performed, not much could have been done with these loops because of the conservative assumptions made on other array reference pairs. Table 7.10 shows these results.

Benchmark	Admis. loops	Loops w sym.	% Loops w sym.	Array pairs	Sym Arr. pairs	% Sym Arr. pairs
alvinn	8	1	13%	8	1	13%
compress	2	1	50%	1	1	100%
ear	24	6	25%	68	28	41%
tomcatv	12	2	17%	37	6	16%
cluster	21	5	24%	97	81	83%
complex	5	1	20%	1	1	100%
asuite	172	7	4%	220	16	7%
csuite	62	1	2%	8	1	13%
clinpack	19	0	0%	15	0	0%
nrcode1	29	0	0%	26	0	0%
nrcode2	10	1	10%	7	1	14%
nrcode3	13	1	8%	9	5	55%
nrcode4	10	0	0%	1	0	0%
heap	1	0	0%	0	0	0%
sort	4	2	50%	23	4	17%
Total	392	28	7%	521	145	37%

Table 7.10: Overall effectiveness of symbolic array dependence analysis.

The following describes the meaning of each column.

Admis. loops: The number of admissible loops.

Loops w sym: The number of admissible loops with array reference pairs requiring symbolic framework for array dependence analysis.

% Loops w sym: The percentage of loops requiring symbolic framework.

Array pairs: The number of array reference pairs collected.

Sym Arr. pairs: The number of array reference pairs containing symbols in index expressions.

% Sym Arr. pairs: The percentage of array reference pairs that contain symbols in index expressions.

7.6.1 Observations

When looking at the numbers for overall effectiveness of symbolic framework, it appears that lack of some support for symbolic array dependence analysis may leave the compiler producing weak code. This observation refers mostly to the results from application benchmarks. Synthetic benchmarks contain many loops which are easy to analyze and thus make the number of loops requiring symbolic array dependence analysis look insignificant. On the other hand real application benchmarks contain many fewer admissible loops with a larger portion of them requiring help from some type of symbolic framework. One can therefore conclude that at some point in the development of an optimizing/parallelizing compiler, symbolic analysis will be required to make it competitive.

Chapter 8

Related Work

There has been considerable research in the area of array dependence testing in general, and in various methods for improving the accuracy of dependence testing. This thesis has addressed the problem of improving the accuracy of array dependence testers by incorporating symbolic analysis. In developing this work, the following three major areas were touched upon:

- Extended SSA Numbering.
- Symbolic Array Dependence Analysis.
- Summary of Array References.

The following subsections describe work related to each of these areas.

8.1 Static Single Assignment Form

There has been a lot of study devoted to developing and perfecting the intermediate representation called Static Single Assignment form [AWZ88, BM94, CFR+89, CFR+91, RWZ88, SG95]. The SSA form has several nice properties, including the fact that each variable use corresponds to exactly one definition point. The SSA form has been widely used in FORTRAN compilers, where pointers are not a problem. Recently there has been increased interest in extending the SSA form to handle the problems associated with pointers.

One of the first steps of accommodating pointers into SSA form, was done by Cytron and Gershbein [CG93]. Their work concentrated on the effects of assignment via pointers. For example, consider an assignment statement such as `*p = 6`.

Depending on where the pointer `p` points to, the assignment may modify different locations. Suppose that `p` at the assignment statement may point to either the variable `a` or `b`. To model the assignment one must take into consideration that both `a` and `b` could be modified.

To do this they devised a well-behaved function (called `IsAlias(p,v)`) taking a pointer `p` and a variable `v`, and it would perform the following: Pointer `p` is tested to see if it actually points to `v`.

- If it does, the function returns the value at `*p`.
- If it does not, the function returns the value at `v`.

After each assignment of the form `*p =...`, a series of statements of the form `v_2 = IsAlias(p,&v_1);` are inserted for each variable that `p` can point to. This has the effect of assigning new names to all variables that can point to `p`.

Their method works fine, but it does not assign versions to pointer variables. Not having versions for pointer variables results in lack of information for statements such as `p = &v;` and pointer references on the right hand side such as `... = *p;`. Recall the problems with such statement from section 3.3.1.

Chow et. al. have developed an SSA form for a language with indirect references [CCL+96]. Their implementation of SSA form lead them to expose *virtual* variables for each pointer. Thus, each assignment via a pointer, `*p = 6` is accompanied by another assignment to `p`'s virtual variable. When the pointer `p` changes the location where it points to, a new virtual variable is assigned to `p`. In addition to virtual variables, their method must expose the read/write sets of statements as special nodes in the program's internal representation. Read nodes are placed before a statement that reads virtual variables, and write nodes are placed after statements writing virtual variables. Thus, their work is quite complex and requires tight coupling of three modules: pointer analysis, read/write analysis, and SSA form.

In contrast, my method of extended SSA numbering is considerably simpler to implement, and can be added to any compiler that has pointer analysis and read/write analysis. There is no need to change the program's intermediate representation since SSA numbers can be stored as just another attribute, or as dataflow information. In practice, it was found that the properties guaranteed by extended SSA numbering are strong enough to support both array dependence testing, and can be used to support several other optimizations (see Appendix A).

8.2 Symbolic Analysis

In the area of symbolic array dependence analysis, the most notable work was by Mohammad R. Haghighat et. al. An overview of their work can be found in [HP96, Hag95]. Unfortunate for the purposes of comparison with this thesis, their work is based on a FORTRAN compiler called Parafrase-2 [PGH+88], which is not exposed to some of the problems introduced by a language like C. However, there are still many aspects that can be compared, which are addressed in this section.

8.2.1 Symbolic Arithmetic

It is clear that performing arithmetic on symbolic expressions is an integral part of any symbolic dependence analysis. Haghighat, in his master's thesis [Hag90], presented a simple representation which I have adopted. It was described in section 4.4.1.

Since then Haghighat et. al. have improved the representation in order to permit the division operation. This required representing each symbolic expression as a quotient of a numerator and denominator. In addition to representing quotient, Haghighat et. al. have built a test of integer divisibility into their analysis. This allows them to determine if the symbolic expression can be divided by an integer. The theory of providing the division operation in symbolic expressions can be found in [HP96] as well as in [DST93].

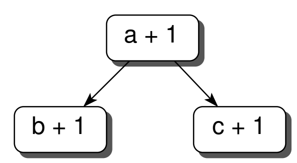
8.2.2 Possible Symbolic Values

Recall from section 4.4 the method used to represent possible symbolic values using PSV trees. For example consider the section of C code in Figure 8.1(a).

```
     if(...)
       a = b;
     else
       a = c;

     e = a + 1;
S93: A[e] = ...
```

(a)



(b)

Figure 8.1: Example symbolic expression with possible values.

At statement *S93* an array dependence analysis is interested in the symbolic expression `e`. The index analysis will expand `e` to be `a + 1`. Next, two possible values for the symbol `a` are modeled by creating two children nodes in the PSV tree, as shown in Figure 8.1(b).

On the other hand, Haghighat et. al. decided not to pursue possible values with such determination. Their approach was to first put the input program in SSA form as shown in Figure 8.2. Their input program can fairly easily be placed in SSA form because the Parafrase-2 is a FORTRAN compiler, instead of C compiler with full pointer capabilities like McCAT.

```
      if(...)
        a_1 = b_1;
      else
        a_2 = c_1;
S94:  a_3 = φ(a_1,a_2);
      e_1 = a_3 + 1;
      A[e_1] = ...
```

Figure 8.2: Example program from Figure 8.1 in SSA form.

Notice that placing the program in SSA form introduced an intermediate symbol `a_3` at statement *S94*. Haghighat et. al. decided to stop their index expression analysis on that symbol. This symbol is considered as a new abstract symbolic value, with its value unknown. The index expression would then remain as just `a_3 + 1`. My results from section 7.3 support this decision. The results show that most PSV trees are built containing only one node. This implies that symbols in the index expression tend to have only one definition point or the alternate definition point forms an induction variable. Therefore, if induction variables are resolved, then the possible inaccuracy introduced by stopping the backward substitution at multiple definitions should produce only a small effect.

To achieve some additional accuracy, Haghighat and Polychronopoulos in [HP96], spend the effort to check the values of a symbol from all joining paths, at control flow joining points. If all the values are the same, then a new abstract value is not created, instead the common value is passed down. For example, in Figure 8.2, a new abstract value is only created if the expressions `b` and `c` cannot be proven equal.

8.2.3 Induction Variables

A well known and proven technique for resolving induction variables is that of Michael Wolfe [Wol92]. This method depends on SSA form. In order to resolve induction variables the input program must first be placed in SSA form. The algorithm identifies induction variables by detecting strongly connected regions in a graph representing the induction variable. The graph is constructed from expression trees with additional edges joining each use of a variable with its definition. The strongly connected region is examined to recognize certain patterns and from that a formula for the induction variable is determined.

The details of this process plus discussion about some more complex induction patterns can be seen in [Wol92]. Let us consider how adaptive this method is to handling symbolic information. Just as any method for resolving induction variables, this method requires detecting loop invariant values. Working with symbolic data just requires the ability to identify loop invariant symbols, and symbolic increments will be accepted. Another issue is symbolic and possibly multiple starting values. Recall that this method first places the input program into SSA form. This guarantees that even if there are multiple starting values they will be joined by one or more ϕ-nodes into a single variable. If this single variable (symbol) is used as the starting value of the induction variable, the formula for the induction variable can be resolved and only then the initial value symbol can be split into several possibilities producing several possible formulas. Notice that an abstract symbol was suggested to be used in resolving multiple starting values. A similar abstract value can be used if the starting value is a symbol of unknown value.

As discussed, the method for resolving induction variables, as developed by Michael Wolfe, can be easily modified to handle symbolic data; however, the implementation of induction variable analysis in Parafrase-2 is different. This implementation, described by Haghighat and Polychronopoulos [HP96], is based on symbolically interpreting the loop several times until a polynomial formula for the induction variable is found. Each variable defined inside the loop is treated as a function of the loop variable. For example, let f represent an induction variable `k` of the loop. $f(0)$ is the value of `k` before the loop starts executing, $f(1)$ is the value of `k` after one loop iteration, and $f(x)$ is the value of `k` after x iterations of the loop.

Now, let us forget that f models an induction variable, and instead treat f as a numeric function. If the values $f(0), f(1), f(2), f(3), f(4)$, are given, *Newton's Advancing Difference Formula* can be used to compute the value of $f(5)$. One simply creates a difference table such as in Figure 8.3 and then extrapolate the table. This can be done because the last difference turned out to be constant implying that f is a polynomial of degree 3. To extrapolate, a 6 is placed in the column labeled $\Delta^3 f(x)$, then the 6 is added to the 14 from column $\Delta^2 f(x)$ to get 20 which is then placed under the 14. Next, the 20 is added to the 30 in column $\Delta f(x)$ to get 50, and lastly the 50 is added to the 61 in column $f(x)$ to get 111 as the value of $f(5)$. The resulting

x	$f(x)$	$\Delta f(x)$	$\Delta^2 f(x)$	$\Delta^3 f(x)$
0	1			
		6		
1	7		2	
		8		6
2	15		8	
		16		6
3	31		14	
		30		
4	61			

Figure 8.3: Difference table for f.

difference table is shown in Figure 8.4

x	$f(x)$	$\Delta f(x)$	$\Delta^2 f(x)$	$\Delta^3 f(x)$
0	1			
		6		
1	7		2	
		8		6
2	15		8	
		16		6
3	31		14	
		30		**6**
4	61		**20**	
		50		
5	**111**			

Figure 8.4: Extended difference table for f.

The coefficients of the polynomial for f can be computed as follows:

$$
\begin{aligned}
f(x) &= \sum_{i=0}^{n} \binom{x}{i} \Delta^i f(0) \\
&= f(0) + x\Delta f(0) + \frac{x(x-1)}{2!}\Delta^2 f(0) + \frac{x(x-1)(x-2)}{3!}\Delta^3 f(0) \\
&= 1 + 6x + \frac{x^2 - x}{2}2 + \frac{x^3 - 3x^2 + 2x}{6}6 \\
f(x) &= x^3 - 2x^2 + 7x + 1
\end{aligned}
$$

Without the availability of arithmetic on symbolic expressions this approach to computing induction variables could not be done. Haghighat and Polychronopoulos generate the symbolic values of each induction variable for a few iterations by just interpreting the loop for a few iterations. They can then construct a difference table and determine the degree of the polynomial for this induction variable. The degree is same as that of the exponent on the Δ operator of the first column with constant values. Constructing a formula for induction variables out of the difference table is then just an arithmetic exercise.

In conclusion, I believe this to be a great contribution of their work, and it may revolutionize the way induction variable analysis is performed.

8.3 Array Reference Summary

In chapter 6, I proposed some applications of the symbolic framework other then in strict array dependence analysis. Leaving the environment of loops, I proposed making a summary of the array references made by each statement. Very similar work has been done some time ago in efforts to make array dependence analysis inter-procedural. In this case, the analysis needs to summarize array references made by a function call. There were three general methods:

Conservative whole array: Any partial reference to an array is considered to access the entire array.

Convex Hull Regions: Multiple array references are summarized by choosing a convex polygon that encompasses all the references.

Atomic reference list: All the individual array references are enumerated and kept separate for the purpose of the summary.

In addition to these three general ideas there are others that propose a more computationally efficient or accurate methods. They are described below.

For the purpose of illustration, consider a function or piece of code that references an array `A` at the following points: `A[1,2]`, `A[4,8]`, `A[10,6]`. Figure 8.5 shows these references graphically. (Source of example is [HK91])

8.3.1 Conservative Method

The conservative method has only one of the two possibilities: "the whole array" or "none of the array". Its effect is illustrated in Figure 8.6.

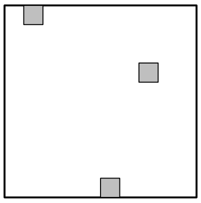


Figure 8.5: Example of three array references.

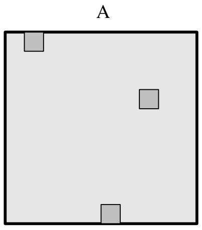


Figure 8.6: Conservative whole array summary.

8.3.2 Convex Hull Regions Method

An other possible way of summarizing several array references is to find a region that encompasses all the array references. The region should be as small as possible and must be able to be expressed using mathematical relations. Triolet, Irigoin and Feautrier in [Tri86] proposed to calculate linear inequalities bounding the set of array location referenced. The result region is effectively the convex hull of the set of points referenced. This is illustrated in Figure 8.7.

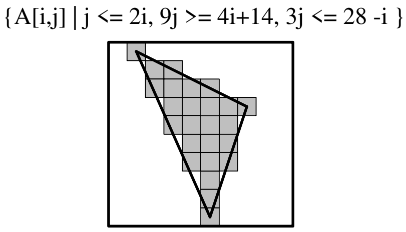


Figure 8.7: Convex hull region summary.

This method is quite precise when selecting a region to summarize a number of array locations referenced. Unfortunately finding the convex hull of a set of points is not a computationally simple task. Furthermore, "finding convex hull of the combined set of inequalities and intersection uses a potentially exponential linear inequality solver [Ban86]" [HK91].

8.3.3 Atomic reference lists

This method can be barely classified as a method for summarizing. Instead of providing a summary of array locations accessed a complete list of all individual array references is provided. This method was developed by Li and Yew and is presented in [LY88]. This method is illustrated in Figure 8.8.

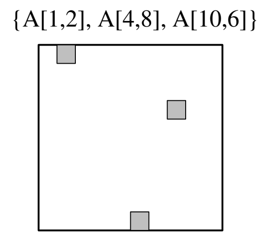


Figure 8.8: Atomic list of array location referenced.

8.3.4 Linearized list

This method is similar to the atomic reference list but the list is presented using linearized array offsets. This method was developed by Burke and Cytron and is presented in [BC86]. This method is illustrated in Figure 8.9.

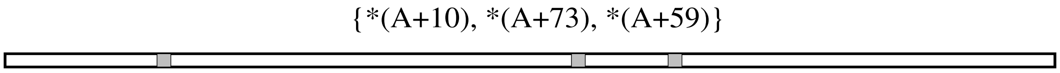


Figure 8.9: Linearized list of array locations referenced.

8.3.5 Restricted Convex Regions

Because finding the convex hull is such an expensive operation Balasundaram and Kennedy in [BK89] proposed to restrict the shape of the region. The region now can have only horizontal, vertical and 45° diagonal borders. This simplifies the inequalities greatly because now the inequalities have only expressions where variables are combined linearly and each variable has a multiplier of only $+1$, 0, or -1. This changes the complexity of intersecting regions to be $O(n^2)$ where n is the dimension of the array. This result comes from the fact that the number of boundaries (inequalities) is exactly $2n^2$.

This method is slightly less precise then the convex hull but it is much simpler computationally. The method is illustrated in Figure 8.10.

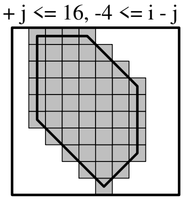


Figure 8.10: Restricted Convex Regions summary

8.3.6 Regular Sections with Bounds and Strides

Thus far when describing a region all the array location within the region were assumed to have been referenced. The method proposed by Callahan and Kennedy in [CK87] changed that by introducing strides. This method is illustrated in Figure 8.11. Computational complexity of this method is similar to that of restricted convex regions but the region is restricted further to exclude diagonal edges making it less precise if the stride is 1.

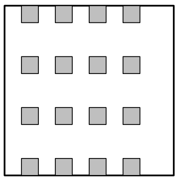


Figure 8.11: Regular Sections with Bounds and Strides summary.

8.3.7 Hierarchical Array Dependence Analysis

The hierarchical array dependence analysis presented in this thesis uses array reference summarization to group array references for compound statements such as conditionals and loops. For array reference summary of a conditional, I used the atomic method, enumerating all the possible array references that could be done by the conditional. For loops, the summary is similar to that of bounds and strides. In the case of admissible loops, each array reference is enumerated with the bounds and strides of loop variables, which appear in index expressions. Note that the stride of a loop variable is always 1 so a reference that accesses every second element is summarized with index expression being `2*i`.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, I presented a framework of symbolic analyses that permit the understanding of array references in the C program under analysis. All of the symbolic analyses were designed and implemented taking into consideration many of the features of the C language including all the uses of pointers. When put together, array dependence analysis is no longer handicapped when presented with array references that make use of variables whose values are not known at compile time.

First, extended SSA numbering was presented. It is an analysis that provides some of the properties of the static single assignment form, which is widely used for compiling FORTRAN programs. In addition, extended SSA numbering took into consideration the uses of pointers and extended some of the properties of SSA form for pointer references. One of the supported properties of SSA form was an important component of the framework helping to define "symbol" in the context of symbolic analyses. In addition to its contribution to symbolic analyses, extended SSA numbering has become a very useful tool for other analyses and transformations. Its simplicity combined with its usefulness make it into a very valuable analysis to have in any compiler.

Next, I presented a new method for representing symbolic expressions for the purpose of accurate and effective comparison with other symbolic expressions. I described how an expression's value should be determined by following reaching definitions and when intermediate values should be preserved. Examples and real programs have demonstrated when and how often accurate representation is important.

In addition to an effective internal representation of symbolic values, induction variable analysis was also examined to consider how it could fit into symbolic array dependence analysis. Some of the considerations were: induction variables incremented by a symbolic loop invariant expression; multiple/symbolic initial values; and, finished loop values for induction variables from loops that have already been executed which are encountered when following reaching definitions.

Reaching the final goal, I presented array dependence analysis in the presence of symbolic index expressions. I suggested a method for trying to remove symbols from index expressions and still use the standard dependence tester for the simplified pairs. This simplification turns out to be powerful and has significant influence. In several cases it is able to simplify complete loop nests into loop nests which don't require a symbolic array dependence tester. In addition to symbolic simplifications, symbolic ZIV and SIV dependence tests were presented which produce useful symbolic results that can be used in parallelizing loops.

As a tangent to the path towards the main goal, I also discussed an analysis called hierarchical array dependence analysis. This analysis provides information about array references within basic blocks and even generates a summary of array references at each hierarchical level of the program. With this information, any two statements at the same hierarchical level and from the same statement list can be analyzed for dependencies due to arrays. The dependency information can be used to generate better code for multi-processor architectures, improve instruction scheduling for unstructured loops, or even as a tool to develop an optimizing transformation such as scalar replacement.

As well as presenting algorithms and methods for the symbolic framework for array dependence analysis, this thesis has evaluated the effectiveness of theses methods using statistics gathering profiler on several real application and synthetic benchmarks. Every component that was developed or modified for the symbolic framework was evaluated and observations about its effectiveness were made. In brief, the results have shown:

- Permitting function calls in loop nests is extremely important. Functions that do not affect the behavior of the loop nest, and don't introduce new array references absolutely must be recognized and accepted by the array dependence analysis. Rejecting loops with such functions can leave an input program with as much as half of its loops unanalyzed.

- Although the additions to induction variable analysis to accept symbolic increments are simple, the generated formulas will require a symbolic dependence test capable of accepting symbolic coefficients for loop variables. Such dependence tests may be difficult to implement. On the other hand, the additions for multiple/symbolic starting values as well as finished loop value induction variables are easy to implement and their output does not require sophisticated dependence tests. In fact, the symbolic components will probably be simplified by symbolic subtraction.

- Building an accurate representation of possible symbolic values can be easily shown to make a difference; however, the need to follow multiple possible definitions appears to be relatively unimportant inside loop nests. This is because most loop index expression are mostly loop variables with possibly a loop invariant symbolic expression common to both instances. On the other hand, accurate representation is much more important in the hierarchical array dependence analysis.

- The actual array dependence testing on symbolic data has shown that the most benefit is to be gained from symbolic simplifications such as symbolic subtraction. These simplifications produce the most accurate dependence information as well as require the least amount of effort to implement. Although, most benefit can be gained from symbolic simplification, symbolic dependence tests need to be provided as well.

In general developing a symbolic framework such as the one presented in this thesis is a very big help to a parallelizing/optimizing compiler. In addition, having presented solutions to overcome problems introduced by pointers, this work can open up many improvements in compilers for languages with multi-level pointers such as C.

9.2 Future Work

Although I have already done much work for this thesis, I am still left with several issues on which I would like to continue work. These issues are itemized below.

- Loop invariant removal. Having examined many loop nests, I have seen that any symbolic analysis has lot to benefit from the well known loop invariant removal optimization. McCAT is not equipped with this optimization yet and I would like to see the effects of this optimization on the performance of the symbolic array dependence analysis. Loop invariant removal may be especially important for compilers such as McCAT that generate intermediate representations such as SIMPLE. These intermediate representation introduce a significant amount of loop invariant code that could be easily removed.

- Additional array dependence tests. Although I have implemented symbolic versions of ZIV and SIV dependence tests, there are still other dependence tests that may be converted to symbolic versions. In addition, the symbolic dependence tests are not capable of analyzing dependence pairs where a symbol multiplies a loop variable. For example, a test pair such as "`x*i`" vs. "`x*i + 1`" cannot be tested because the loop variable `i` is multiplied by a symbolic expression `x`. The question which I would like to address is "Can such a test pair be identified as a SIV-strong pair and effectively tested for dependencies?"

- Index expression analysis had a very specialized purpose. It was only intended to generate an effective representation of array index expressions. In general, array index expressions are not numerous, and therefore the analysis was permitted to be detailed. Can the index expression analysis be restructured to be a dataflow analysis where each expression of the program under analysis can be quickly expressed by a detailed PSV tree or a similar representation?

- Hierarchical array dependence analysis. There are some applications of the hierarchical array dependence analysis which have not been implemented and their effectiveness not yet studied. Studying the possibilities of this analysis are certainly interesting and worth while to work on.

Appendix A

Other applications of Extended SSA Numbering

Besides the main goal for which extended SSA numbering was designed, extended SSA numbering has several other applications. These applications are mostly to support other analyses or provide vital information to optimizing transformations. Here are some analyses and transformations that use SSA numbers in McCAT.

A.1 Induction variables

Wolfe [Wol92] has presented a method for recognizing induction variables and computing their formulas. His algorithm used SSA form. If one does not have SSA form, or if the programming language one is analyzing makes SSA form difficult to implement, then to recognize induction variables one must recognize when the pair (variable, set of reaching definitions) have been already visited when performing backward substitution. Properties 3 and 4, as defined in section 3.5, can perform this test without looking at the set of reaching definitions.

A.2 Recognition of `ref` parameters

In C there exists no syntax for passing parameters by reference; thus, to accomplish this, programmers use pointers. For the purposes of program understanding, it is beneficial to know which pointers are really `ref` parameters. Using extended SSA numbering, `ref` parameters can be easily spotted by comparing the primary SSA number of each formal pointer parameter before and after applying extended SSA numbering to the body of a function. If the primary SSA numbers remain the same, the pointer has never changed the place it points to; thus, it can be treated as a `ref` parameter.

A.3 Strengthening reaching definitions

Consider the example in Figure A.1. The key point of interest is the possible definitions of `*p`. When analyzing statement *S97*, variable `n` is found to be defined at *S95* and variable `m` is defined at *S96*. Statement *S97* introduces a new definition, but it is not known which variable it is defining until run time. It is possible that a new definition for `n` or for `m` is being introduced. Because it is not known which one, it is necessary to be conservative and say that from now on the possible definitions for `n` are {*S95*, *S97*} and for `m` they are {*S96*, *S97*}. Then at statement *S98* the reaching definitions for `*p` are the union of the reaching definitions for `n` and `m` which is {*S95*, *S96*, *S97*}.

```
      if(...)
        p = &n;
      else
        p = &m;

S95:  n = 1;
S96:  m = 2;
S97:  *p = 3;
      /* Cannot kill defs S95, S96 */

S98:  x = *p + 1;
      /* possible def of *p = {S95,S96,S97} */
```

Figure A.1: Example of reaching definitions for pointer references.

Now consider using extended SSA numbering to help. If SSA#(`p`,*S98*) = SSA#(`p`,*S97*) and ExtSSA#(`p`,*S98*) = ExtSSA#(`p`,*S97*) then the value returned by `*p` at statement *S98* will be the same as the one produced by statement *S97*. It can therefore be concluded that the sole reaching definition is *S97*.

A.4 Reduction of Indirect References

From the conclusions of section A.3, it is possible to come up with the following optimizing transformation. If a definition via indirection is followed by uses of the same pointer dereferenced with matching primary and secondary SSA numbers, a scalar temporary variable is introduced to pass the value without indirection to all the uses. Similarly if more than one indirect reference via the same pointer such that both primary and secondary SSA numbers match is found, the referenced value is stored in a temporary variable to be accessed directly in the future matching uses. The two transformations are illustrated in Figure A.2. Care also has to be taken to ensure that the statement at which the value is stored into a temporary variable dominates[1] all the matching uses being replaced.

$*p_{a,b}$ = expr;		temp = expr;
		$*p_{a,b}$ = temp;
...	=>	...
x = $*p_{a,b}$;		x = temp;
...		...
y = $*p_{a,b}$;		y = temp;
x = $*p_{a,b}$;		temp = $*p_{a,b}$;
		x = temp;
...	=>	...
y = $*p_{a,b}$;		y = temp;
...		...
z = $*p_{a,b}$;		z = temp;

Figure A.2: Two transformations to reduce indirect references.

I have applied this transformation to many benchmarks. Unfortunately, with many of the benchmarks, there was no opportunity to apply the transformation. Table A.1 gives a list of benchmarks and their size in number of SIMPLE statements, where the transformation was applied. Table A.2 shows the effectiveness of the transformation. Each benchmark was examined to see the effect of the transformation. First, the number of times the transformation was applied was counted. Then for

[1]Statement S dominates statement T if and only if S is in every execution path from the $START$ to T

each of the indirect references removed a counter increment was inserted into the code. Running this transformed program then printed this counter showing how many indirect references were actually removed when the compiled program is run on sample data. The actual run time savings in terms of time was negligible on a fast processor with cached memory; however, on a distributed memory machine where memory access is not necessarily a local operation may turn out to be significant. Because of this reason, speedup information is not presented.

Name	Description	Size
vlsi	VLSI chip testing problem	513
frac	Represents floating point into a quotient of two integers	126
chomp	Game solver using game tree	467
queens	N Queens problem	333
stanford	Stanford baby benchmark suite from John Hennessy	1195
travel	Traveling Salesman problem	450
puzzle	Solve 15 numbered square puzzle	366

Table A.1: List of benchmarks to which we applied the indirect reference reduction.

Benchmark	Num Transf.	Ind. Ref. Savings
vlsi	2	1
frac	7	25
chomp	4	138
queens	5	12,513
stanford	2	38,450
travel	1	67,037
puzzle	1	339,579

Table A.2: Results of applying our indirect reference reduction transformation.

The effectiveness of this transformation varied greatly. For example, two benchmarks, `frac` and `vlsi` produced very little benefit at run time, even though the benchmark `frac` applied the transformation more times than any other. On the other hand, a benchmark like `puzzle` produced the most benefit at run time, even though there was only one place in the program where the transformation could be applied.

Appendix B

Decoding the McCAT ADT output

The McCAT array dependence tester generates detailed output for each admissible loop. To help users of the McCAT array dependence tester read the information, I present this brief description of the output. For the purpose of explaining each component please refer to Figure B.1.

```
       for ( i/*1*/ = 0 ; (i/*2*/ <= 99) ; i/*3*/ = (i/*2*/ + 1) )
       {
/*S1*/   A/*2*/[i/*2*/] = 5;
         temp_14/*2*/ = (i/*2*/ + x/*1*/);
/*S2*/   A/*3*/[temp_14/*2*/] = 7;
       }

/**************************************
Summary of Array Dependence Information
***************************************
Loop variable : i(2)   Low-bound : 0 -- Upper-bound : 99


Possible Dependence Info between :S1 - S2 are :
The index PSV for LHS array reference are:
    Dimension 1:
01: + 1*i(2)
The PSV for RHS array reference is:
    Dimension 1:
01: + 1*i(2) + 1*x(1)

     Possible Canonical Pair : 1
Original Canonical pairs
LHS_Subscript :  0 + 1*i(2)  RHS_Subscript :  + 1*x(1) + 1*i(2)
Subscript-pair type : SURE-PAIR
Dependence-pair type : FLOW Dependence
Result : SURE-DEPENDENT with Distance Vector : [ - 1*x(1)]
                        Dependence Vector : [*..*]
Dimension 0 -  Dependence tester :  Symbolic SIV_Strong test
```

Figure B.1: Example output from McCAT symbolic array dependence analysis.

B.1 Generated C dump

When compiling a C program, McCAT generates an output file that is an equivalent C program with optimizing and simplifying transformations applied. This generated file is called the C-dump.

B.1.1 Extended SSA Numbering

In the C-dump, each variable use is accompanied by its SSA numbers. The SSA numbers are printed as comments immediately following each variable use. For example, in Figure B.1 the instance of the variable `i` in the initializer of the for loop has SSA number 1. In the case of indirect reference, the pointer is accompanied by a comment with two numbers separated by a comma.

B.1.2 Statement labels

Statements that contain array references are labeled to be identified in the summary of array dependencies. In Figure B.1 the two statements that have array references were labeled `/*S1*/` and `/*S2*/` .

B.2 Summary of array dependence information

After each admissible loop nest a long comment is inserted containing the summary of all the dependencies due to array references. It contains a short summary of the loop variables, and for each pairing of arrays, the list of dependencies that it generates.

B.2.1 Loop variable summary

Each loop variable is described by a single line at the very top of the summary. The name of the variable with its SSA number, given in round braces, is first followed by the lower and upper bounds. The SSA number of the loop variable is the SSA number of the variable in the loop body, not in the initializer or the incrementer.

The lower bound will always be 0, because loops are first normalized to start at 0 and be incremented by 1. The upper bound may be a number, a range of numbers, an asterix or a symbolic expression. An upper bound written as a number implies that the loop variable ranges from 0 to that number. If the upper bound is given as a range of numbers then the upper bound is unknown but GCP resolved the upper bound to a numeric range. If GCP cannot resolve an upper bound to a range, an asterix is placed as the upper bound implying that the upper bound is not known. In the process of symbolic dependence testing, the upper bound is sometimes needed as a symbolic expression. For loops where the upper bound was needed, a symbolic expression may appear there.

B.2.2 Possible dependence info

For each pair of array references where the array name or aliasing information leads to a possible dependence a section is written starting with a line containing "`Possible Dependence Info between:`". This line also states the two statements involved using their labels.

Next the PSV trees for each index expression of this array reference are displayed. There is a PSV tree for both array references and for each dimension of the reference.

PSV trees

PSV tress are output in such a way as to make them most readable and possibly displayable by a utility program called `xtree`.

The first component of PSV trees is a number on the very left of each line. In the example of Figure B.1 the number `01:` is present. This number is the level of the tree node in the PSV tree. This number is there so more complex PSV trees can be displayed by a utility program called `xtree`. To view a PSV tree cut the lines printing the tree out into a file and give the following command.

```
xtree -iformat number < PSV-file
```

The program `xtree` is not graph viewing program, but a user interface utility. Because of this a postscript file is not generated. The way to print the tree can only be done by grabbing the contents of a window using the `xwd` utility.

After the number giving the level of the PSV node, follows the symbolic expression of the node. Each symbol in the node has its SSA numbers printed in round braces and possibly a range of values provided by GCP in square braces.

Figure B.2 shows a larger PSV as it is output by McCAT and its graphical representation. Figure B.2 also shows the `xtree` utility displaying the PSV tree.

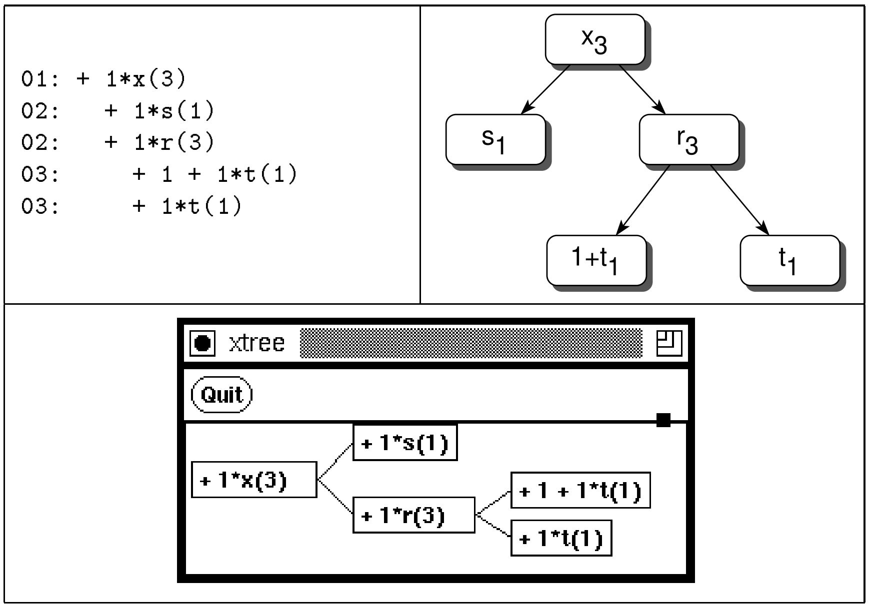


Figure B.2: Example PSV as output by McCAT and its graphical representation.

Possible Canonical Pairs

Recall from section 5.1, that the algorithm for finding all the dependencies of an array pair stops when it is satisfied with all the results in a plan line. The results of the tests in the satisfactory plan line are listed as possible canonical pairs. They are numbered, and in the example from Figure B.1 there is only one possible canonical pair numbered 1.

Each possible canonical pair contains the following information:

- Index expressions in canonical form for both sides and each dimension.
- The type of dependence due to same array name or aliased arrays. This information is labeled "Subscript-pair type".
- Whether the dependence is an output, or a flow/anti dependence.
- The result of the dependence test. This portion contains distance vectors which may contain symbolic expressions and loop variables. The loop variables are labeled "`{left}`" or "`{right}`" to identify which instance of the loop variable is being referred.

- The dependence test that was applied. For each dimension that was tested, the name of the dependence test that was applied is given.

Bibliography

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[Ban86] Utpal Banerjee. Direct parallelization of call statements–a review. CSRD Rpt. No. 576, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1986.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. pages 162–175, June 25–27, 1986. *SIGPLAN Notices,* 21(7), July 1986.

[BDH+87] Mark Byler, James R. B. Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple version loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 312–318, St. Charles, Illinois, August 17–21, 1987.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Computers*, pages 746–757, October 1966.

[BK89] Vasanth Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 41–53, Portland, Oregon, June 21–23, 1989. *SIGPLAN Notices,* 24(7), July 1989.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, New York, June 20–22, 1990. *SIGPLAN Notices,* 25(6), June 1990.

[CCL+96] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. An implementation of interprocedural bounded regular section analysis. In *Proceedings of the 6th International Conference on Compiler Construction, CC '96*, Lecture Notes in Computer Science, pages 253–267, Linkoping, Sweden, April 22–26, 1996. Springer-Verlag.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 36–45, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[CK87] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Conference Proceedings, 1987 International Conference on Supercomputing*, number 297 in Lecture Notes in Computer Science, Athens, Greece, June 8–12, 1987. ACM, Springer-Verlag.

[Don94] Christopher M. Donawa. The design and implementation of a structured backend for the McCAT C compiler. Master's thesis, McGill University, Montréal, Québec, March 1994.

[DST93] J. H. Davenport, Y. SIRET, and E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation.* Academic, London, 2nd ed. edition, 1993.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[EH94] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, Toulouse, France, May 16–19, 1994. IEEE Computer Society Press.

[Ero95] Ana Marie Erosa. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, McGill University, Montréal, Québec, May 1995.

[Fea91] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices,* 26(6), June 1991.

[GSW95] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[Hag90] Mohammad Reza Haghighat. Symbolic dependence analysis for high performance parallelizing compilers. Master's thesis, University of Illinois at Urbana-Champaign, May 1990.

[Hag95] Mohammad R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Publishers, August 1995.

[HDE+92] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[HK91] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on on parallel and distributed systems*, 2(3):350–360, July 1991.

[HMT+95] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[HMT+96] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[HP96] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

[HTZ+96] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[JH94] Justiani and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In Peter A. Fritzson, editor, *Proceedings of the 5th International Conference on Compiler Construction, CC '94*, number 786 in Lecture Notes in Computer Science, pages 309–323, Edinburgh, Scotland, April 7–9, 1994. Springer-Verlag.

[Jus95] Justiani. A practical array dependence analysis for an optimizing/parallelizing c compiler. Master's thesis, McGill University, Montréal, Québec, July 1995.

[LY88] Zhiyuan Li and Pen-Chung Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages and Systems*, pages 85–97, New Haven, Connecticut, July 19–21, 1988. ACM SIGPLAN. *SIGPLAN Notices,* 23(9), September 1988.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 1–14, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices,* 26(6), June 1991.

[PGH+88] Constantine Polychronopoulos, Miland Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: A new generation parallelizing compiler. CSRD Rpt. No. 837, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1988.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, San Francisco, California, January 22–25, 1995.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.

[Tri86] Remi Triolet. Interprocedural analysis for program restructuring with Parafrase. CSRD Rpt. No. 538, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, August 1986.

[VCH96] Clark Verbrugge, Phong Co, and Laurie J. Hendren. Generalized constant propagation a study in c. In *Proceedings of the 6th International Conference on Compiler Construction, CC '96*, Lecture Notes in Computer Science, pages 74–90, Linkoping, Sweden, April 22–26, 1996. Springer-Verlag.

[Wol92] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.