PUTTING POINTER ANALYSIS TO WORK

by
Rakesh Ghiya

School of Computer Science
McGill University, Montreal

May 1998

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Copyright © 1998 by Rakesh Ghiya

Abstract

Pointer analysis has recently been a subject of active research. The focus of most techniques is on: (1) estimating the targets for stack-directed pointers, (2) computing relationships between heap-directed pointers, or (3) developing faster or more space-efficient algorithms. However, little attention has been paid to actually using the results of pointer analysis for compiler optimizations. This thesis addresses the problem of how to apply pointer analysis information to a wide variety of compiler applications. That is, once the pointer relationships are computed, what can we do with this information? How do we put pointer analysis to work?

We have used the pointer analyses implemented in the framework of the McCAT C compiler as the basis of our study. These analyses include, points-to analysis for stack-directed pointers, and connection and shape analyses for heap-directed pointers. We demonstrate that the fundamental problem is that one must be able to compare the memory locations read/written via pointer indirections, at different programs points, and one must also be able to summarize the effect of pointer references over regions in the program. It is straightforward to compute read/write sets for indirections involving stack-directed pointers using points-to information. However, for heap-directed pointers we show that one needs to introduce the notion of *anchor* handles into the connection analysis and then compute read/write sets to the heap with respect to these anchor handles.

Based on the read/write sets we show how to extend traditional optimizations like common subexpression elimination and loop-invariant removal to include pointer references. In the same vein, we propose a novel optimization called location-invariant removal. We also demonstrate the use of our information on more advanced techniques such as array dependence testing and program understanding. We have implemented our techniques in the McCAT C compiler, and we demonstrate examples of applying

our methods on a set of pointer-intensive C benchmarks, as well as present concrete empirical data on the improvements achieved.

We also present techniques to use pointer analysis information for detecting parallelism in C programs with recursive data structures. In addition to the read/write sets, we use shape analysis information and data structure access paths for dependence testing. The dependence tests are specifically tailored for detecting three parallelism patterns that frequently arise in C programs. These patterns include, function calls that access disjoint sub-pieces of tree-like data structures, pointer-chasing loops that traverse list-like data structures, and array-based loops which operate on arrays of pointers pointing to disjoint data structures. To identify loop parallelism, we introduce special tests for identifying loop-carried dependences in the context of recursive data structures. We have implemented the tests in the framework of the McCAT C compiler, and present experimental results for a set of C benchmark programs to evaluate their effectiveness.

Résumé

L'analyse de pointeurs est depuis quelques temps le sujet de recherches actives. Ces recherches ont pour but: (1) de déterminer les adresses auxquelles les pointeurs se réfèrent, (2) de trouver les liens entre pointeurs au heap, et (3) de développer des algorithmes plus rapides et utilisant moins d'espace. Par contre, l'application de cette analyse à l'optimisation où à d'autres applications n'a reçu que peu d'attention. Dans la thèse présente, nous adressons ce problème.

Les analyses implantées dans le cadre du compilateur C "McCAT" sont utilisées; celles-ci comprennent, pour les pointeurs à la pile, l'analyse "pointe-à", et pour les pointeurs au heap, les analyses de connectivité et de forme. Nous montrons que le problème principal consiste à comparer les adresses mémoire auxquelles le programme accède au travers d'indirections, ansi qu'à extraire les caractéristiques de références mémoire dans des régions données du programme. Pour les pointeurs à la pile, il est facile de construire les ensembles de lecture/écriture à l'aide de l'analyse "pointe-à". Par contre, pour les pointeurs au heap, il faut introduire la notion d'alias ancré à l'analyse de connectivité, et construire les ensembles de lecture/écriture à partir de ces alias ancrés.

A l'aide des ensembles de lecture/écriture, nous étendons les optimisations traditionnelles telles l'élimination de sous-expressions communes et l'extraction d'invariantes de boucles, à l'inclusion de références pointeurs, et introduisons une optimisation, l'extraction d'invariantes à la position. Nous montrons aussi l'utilité de nos techniques à des applications avancées, telles la compréhension de programmes et l'examen de dépendance des tableaux. Ces techniques, implantées dans le cadre du compilateur McCAT, sont appliquées à des programmes C où les pointeurs sont utilisés intensivement, et nous présentons des résultats prouvant les progrès réalisés.

Finalement, nous rendons possible la détection de parallélisme dans des programmes C aux structures de données récursives, grâce à l'analyse de pointeurs. Les

ensembles de lecture/écriture, ainsi que l'analyse de forme et les chemins d'accès de données, sont utilisés dans l'examen de dépendances. Les examens de dépendance détectent trois types spécifiques de parallélisme, fréquents en C: l'appel de fonctions accédant à des portions disjointes d'une structure en arbre, la boucle parcourant une liste linéaire chainée, et la boucle parcourant un tableau de pointeurs à des structures disjointes. Nous introduisons des tests mettant en évidence des dépendances, dans les boucles, se rapportant aux structures de données récursives. Ces tests, implantés dans le cadre du compilateur McCAT, sont appliqués à des programmes C, et les résultats expérimentaux sont presentés.

Acknowledgments

First thanks go to my advisor Laurie Hendren, who has been very supportive and encouraging all through my stay at McGill. Her caring approach, contagious cheerfulness, and confidence in my abilities have done much to keep my morale high. Her willingness to listen to my ideas prompted many stimulating discussions, which greatly helped me in gaining clear insights and keeping focused. She has also generously supported my studies and saved me from many a worry. Working with her has been a truly rewarding and pleasant experience, which I will always cherish.

I would also like to thank Professor Guang Gao and Professor Prakash Panangaden, from whose foundational courses at McGill, I have learned to appreciate the implications of architectural issues for compilers, and the theoretical underpinnings of programming languages.

This thesis has built upon the work of many people who have worked on the McCAT compiler. Special thanks to Bhama Sridharan, Maryam Emami, Justiani, Ana Erosa, Chris Donawa, Morris Bernstein and Chris Lapkowski.

The one year stint at Siemens Labs in Munich before starting PhD studies at McGill, greatly helped me in chalking out my directions. Thanks to my very helpful colleagues, Thomas Henties and Claus-Peter Wiedemann. The fun friends in Munich: Martin, Farida, Hermine, Susanne and Adi, and Judy and Owain, made my stay a very pleasant one. I'll always cherish the fun times in the 'biergartens'.

During my stay in Montreal, I have been lucky to meet many nice friends, who have added much color to life and whom I owe many thanks. Specially to: Yingchun Zhu for her friendship, company and support, and the many interesting discussions during coffee breaks; Nasser Elmasri for the many delightful chats; Andres Marquez for keeping things light and lively; Herve Avril for the absorbing Saturday afternoon

snooker 'marathons'; Alain Turki for organizing the many wonderful weekend outings and bringing the lot together; Nilanjan Mukherjee for all his help; Joe Hummel for the jolly company on the conference circuit; and Luis Lozano, Erik Altman, and V. C. Sreedhar for their invaluable help in making a career choice.

The administrative staff in the school has provided excellent support. Special thanks to Franca Cianci and Lise Minogue, for making life a lot easier.

Finally, I would like to thank my parents for their support in all my endeavors, and my sister Renu and my cousin Vivek for the many light-hearted phone chats that have always been a welcome diversion.

This work was partly supported by NSERC and FCAR.

To my parents

And

To Uncle S.N.

Contents

Abstract ii

Résumé iv

Acknowledgments vi

1 Introduction and Motivation 1

1.1 Pointer Analysis – An Introduction 2

1.2 Pointer Analysis – A Historical Perspective 4

1.2.1 Stack Pointer Analysis 4

1.2.2 Heap Pointer Analysis 8

1.3 Pointer Analysis in the McCAT C Compiler 15

1.4 Using Pointer Analysis 17

1.4.1 Pointers and Compiler Optimizations 17

1.4.2 Computing Read/Write Sets 21

1.5 Thesis Goals and Organization 23

2 Setting **25**

2.1 The McCAT C Compiler 25

2.2 SIMPLE Intermediate Representation 28

2.3 Points-to Analysis 30

2.4 Interprocedural Analysis Framework 31

2.4.1 Representing Calling Contexts 34

2.4.2 Map Information in the Invocation Graph 36

2.4.3 Resolving Function Pointers 38

2.5 Computing Stack Read/Write Sets 40

2.6 Heap Analyses 41

3 Connection Analysis and Heap Read/Write Sets **45**

3.1 Connection Analysis 46

3.1.1 Illustrative Examples 48

3.2 Computing Read/Write Sets Based on Connection Analysis 53

3.3 Dropping Anchors 53

3.4 Introducing Anchors in Connection Analysis 57

3.4.1 Generating Anchor Handles 57

3.4.2 Augmented Connection Analysis for Basic Statements 59

3.4.3 Augmented Connection Analysis for Control Statements . . . 62

3.4.4 Augmented Connection Analysis for Function Calls 63

3.5 Empirical Evaluation of the Analyses 70

3.6 Understanding Anchor Handles 72

3.7 Summary 76

4 Applications **77**

4.1 Scalar Optimizations 78

4.1.1 Loop Invariant Removal 80

4.1.2 Location Invariant Removal 83

4.1.3 Common Subexpression Elimination 85

4.2 Experimental Results for Scalar Optimizations 89

4.2.1 Optimizations Applied with Stack and Heap Read/Write Sets 90

4.2.2 Benefits of Using Heap Read/Write Sets 92

4.2.3 Runtime Improvements with Stack and Heap Read/Write Sets 97

4.3 Performance Improvement for EARTH-C Programs 102

4.4 Improving Array Dependence Tests 104

4.5 Program Understanding/Debugging 106

4.6 Other Applications 112

4.6.1 Detecting Run-Time Constants 112

4.6.2 Compiler-Directed Prefetching 113

4.6.3 Identifying Memory Leaks 115

4.7 Related Work on Using Pointer Analysis 115

5 Detecting Parallelism in Programs with Recursive Data Structures 119

5.1 Parallel Patterns 120

5.2 Shape Analysis 124

5.2.1 Illustrative Examples 125

5.3 Identifying Parallelism 129

5.4 Function Call Parallelism 130
5.4.1 Shape Dependence Test 134
5.5 Loop Parallelism 146
5.5.1 Good Loop Detection 148
5.5.2 Verifying the Navigator 154
5.5.3 Detecting Heap-based Loop-carried Dependences 155
5.5.4 Overall Strategy 159
5.5.5 Identifying *forall* Loops 160
5.6 Experimental Results 161
5.7 Related Work 164
5.8 Summary 165

6 Conclusions and Future Work **166**

6.1 Future Work 169

Bibliography **172**

List of Figures

1.1 The Pointer Classification 3

1.2 Example Program for Heap Analysis 5

1.3 An Example of 2-limiting 10

1.4 Motivating Examples 19

1.5 Example for Computing Read/Write Sets 22

2.1 The McCAT Compiler 27

2.2 List of the 15 Basic SIMPLE Statements. Variables `x` and `y` denote `varname`. Variables `a`, `b`, and `c` denote `val`. Variables `p` and `q` denote `ID`. 28

2.3 SIMPLE Grammar for a `varname` 29

2.4 Simplification of an Indirect Reference 29

2.5 An example for Points-to Analysis 32

2.6 Points-to Information and Heap Analyses 33

2.7 Invocation Contexts 35

2.8 Invocation Graph for Recursion 35

2.9 Invocation Graph for Mutual Recursion 36

2.10 Map Information 37

2.11 Invocation Graph for Function Pointers 39

2.12 An Example Showing Various Heap Abstractions 43
3.1 An Example Connection Matrix 47
3.2 Demonstrating Connection Analysis 50
3.3 Computing Connection Matrix C_n from C for Basic Statements . . . 51
3.4 Interprocedural Startegy 52
3.5 Example for Connection-based Read-Write Sets 54
3.6 Tricky Example 55
3.7 Dropping Anchors and Inserting Ghost Copy Statements 56
3.8 Augmented Connection Analysis For Basic Statements 60
3.9 Computing Heap Read/Write Sets for Basic Statements 61
3.10 Augmented Connection Analysis for Control Statements 62
3.11 Augmented Connection Analysis for Call Statements 64
3.12 Example for Computing Function Call Read/Write Sets 65
3.13 Step 1 for Computing Heap Read/Write Sets for Function Calls . . . 67
3.14 Step 2 for Computing Heap Read/Write Sets for Function Calls . . . 69
4.1 Illustrating Scalar Optimizations 79
4.2 Loop Invariants 81
4.3 Removing Loop Invariants 84
4.4 Location Invariants 85
4.5 Removing Location Invariants 86
4.6 Eliminating Common Subexpressions 87
4.7 Computing Available Expressions 88

4.8 Examples for Loop Invariants (LIR) and Location Invariants (LcIR) 93

4.9 Common Subexpression Elimination (CSE) Examples 94

4.10 The Framework for Evaluating the Benefits of Scalar Optimizations . 99

4.11 Using Scalar Optimizations for Optimizing EARTH-C Programs . . . 103

4.12 Using Read/Write Sets for Program Understanding and Other Applications 107

4.13 Browser Snapshot showing Stack R/W sets 109

4.14 Browser Snapshot showing Heap R/W sets 110

4.15 Browser Snapshot showing Field-sets 111

4.16 Cache Example for Runtime Code Generation 114

4.17 Example Illustrating Prefetching for Recursive Data Structures . . . 116

4.18 Identifying Memory Leaks 117

5.1 Parallelism Patterns 121

5.2 Staggered Execution of a *foreach* Loop 122

5.3 Converting a *foreach* Loop to a *forall* Loop 123

5.4 Testing for shape 123

5.5 Example Direction and Interference Matrices 126

5.6 Example Demonstrating Shape Estimation 127

5.7 Estimating Shape with *accessibility* Criterion 128

5.8 Acyclicity of Dag Data Structures 128

5.9 Data Dependences 130

5.10 Checking if two Statements are Dependent 131

5.11 Using Read/Write Sets to Check Dependence 132

5.12 Illustrating Hierarchical Dependence Testing 133
5.13 Using Shape Analysis for Dependence Detection 135
5.14 Collecting Pointers that Access the Heap 136
5.15 Computing Definition Chains 137
5.16 Function-call Parallelism 138
5.17 Using Definition Chains to Find the Anchor and the Access Paths . . 140
5.18 Checks for Field Updates 141
5.19 Comparing Access Paths for Dependence Detection 142
5.20 Illustrating Access Path Comparisons 144
5.21 Example *foreach* Loop 147
5.22 Rules for Identifying the Navigator 150
5.23 Computing Definition Chains for Loop-resident Definitions 152
5.24 Identifying the Navigator 153
5.25 Test for Loop-carried Heap Dependences 156
5.26 Loop-carried Dependencies and Shape Attribute 158
5.27 Example *forall* Loop 160
5.28 *forall* and *foreach* Loops from Benchmark Codes 163

Chapter 1

Introduction and Motivation

Optimizing and parallelizing compilers rely upon accurate static disambiguation of memory references i.e. determining at compile-time, if two given memory references always access disjoint memory locations. Unfortunately the presence of aliases in programs makes memory disambiguation a non-trivial issue. An alias arises in a program when there are two or more distinct ways to refer to the same memory location. In the presence of aliasing, two seemingly dissimilar references can access the same memory location. Program constructs that introduce aliases are arrays, pointers[1] and pointer-based dynamic data structures. For example, the array references `a[i+2*j]` and `a[j+2*i]`, the pointer dereferences `*q` and `*p`, and the structure accesses `p->item` and `q->next->item`, can lead to the same memory location.

Over the past twenty years, powerful data dependence analyses have been developed to resolve the problem of array aliases [Ban88, Wol89, ZC90]. These analyses use integer programming techniques to determine if two array subscript expressions can evaluate to the same value. They form the core of present day optimizing/parallelizing compilers. The problem of calculating pointer-induced aliases, termed *pointer analysis*, has also received significant attention from the research community over the past few years [LR92, CBC93, EGH94, WL95, Ruf95, Ste96, ZRL96, SH97b]. This emphasis has mainly stemmed from the increasing popularity of languages supporting pointers such as C, C++ and Fortran90. Further emphasis comes from application areas which primarily use pointer-based dynamic data structures.

[1] Call-by-reference parameters can be considered as a restricted case of pointer usage.

Important examples include: computational fluid dynamics, computational geometry, computational biology, computer graphics, N-body and circuit simulations. Currently, pointer analysis technology is poised to be incorporated into production compilers.

Most of the current research on pointer analysis focuses on developing faster and/or more precise algorithms. While this is certainly a laudable goal, another equally important aspect of the problem is actually *putting the pointer analysis to work*. That is once the information from pointer analysis is available, how it can be used for more accurate memory disambiguation and how much this improved accuracy impacts performance. This is the focus of this thesis.

In the rest of this chapter, we first illustrate the pointer analysis problem in section 1.1. We discuss related work on this topic in section 1.2. In section 1.3, we present the McCAT [HDE+93] pointer analysis framework that is used in this thesis. In section 1.4, we identify the missing links between pointer analysis and its use for program optimization. Finally in section 1.5, we briefly describe our approach to this problem, and present the goals and contributions of this thesis.

1.1 Pointer Analysis – An Introduction

In order to understand how to use pointer analysis, one must first understand pointer analysis itself. Pointer analysis can be divided into two distinct subproblems. The first subproblem focuses on pointers pointing to statically-allocated memory objects (typically on the stack). These are called *stack-directed* pointers. The second subproblem deals with *heap-directed* pointers, which point to objects dynamically allocated in the heap. A pointer pointing to an object, implies that it contains the memory address of the given object. For example in Figure 1.1, pointer `p` points to the object `x`. Further, pointer variables `p` and `q` are stack-directed, while `r` and `s` are heap-directed. Note that the pointer variables themselves are resident on the stack. Also, a pointer can fall into both categories, if it can possibly point to locations on the stack as well as in the heap.

Stack-directed pointers exhibit the important property that their targets always possess a name. This is because all data objects allocated on the stack have compile-time names. Using this property, alias information for such pointers can be accurately

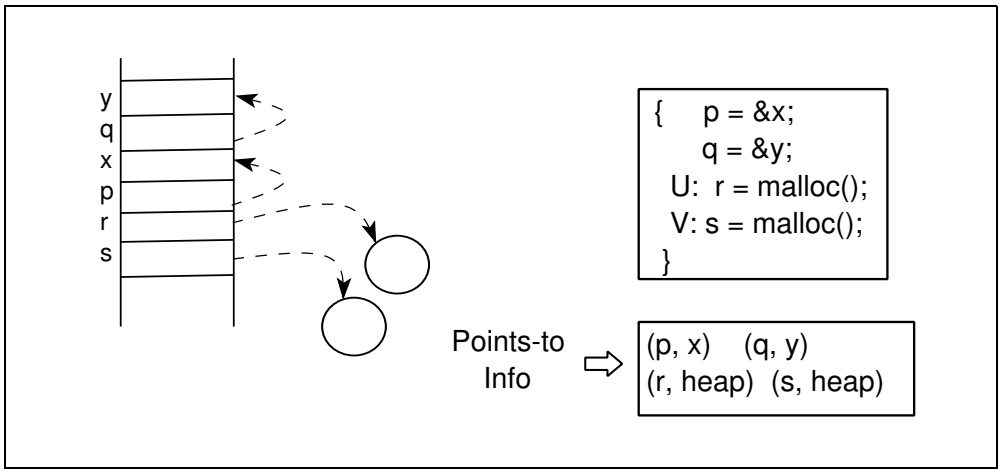


Figure 1.1: The Pointer Classification

captured in the form of *points-to* pairs. For example in Figure 1.1, we have points-to pairs `(p, x)` and `(q, y)` denoting that pointer variable `p` points to the data object `x` and pointer variable `q` points to the data object `y`. This approach was initially proposed by Emami et. al [EGH94] and is used by most of the recent pointer analysis algorithms [Ruf95, WL95, Ste96, SH97b].

Unfortunately this nice property does not hold for heap-allocated data items. In fact all the objects in the heap are *anonymous*. They can be accessed only through pointer dereferences like `*r`, `r->item` or `a[i]`, where `r` and `a` are heap-directed pointers. One cannot use a simple naming scheme to name heap objects, as a potentially infinite number of them can be created. Further, objects in the heap are dynamically linked, and more importantly delinked. Hence, there is no natural way of naming even collections of objects (e.g. linked structures). Unlike arrays, both the number of linked structures and the number of objects belonging to a given linked structure vary dynamically.

There have been two approaches to the heap analysis problem. The first approach has been to associate heap objects with some set of static names, and use the same framework for heap analysis as for stack analysis. The naming schemes proposed are as simple as using one name, *heap*, to represent all heap objects [EGH94]. More precise schemes name objects using their allocation site [LR92], or allocation site prepended with the procedure string representing the given calling context [CBC93, WL95].

For the code fragment in Figure 1.1, the first naming scheme gives the points-to pairs `(r, heap)` and `(s, heap)`. Using allocation sites instead, would give the pairs `(r, malloc_U)`, `(s, malloc_V)`.

However, treating the heap as a set of static named locations can lead to significant imprecision, as one name can represent several heap objects which may be completely unrelated [Deu92]. Another approach in this regard has been to develop special *storeless heap analyses* which provide more accurate information. In contrast to *store-based* techniques like points-to analysis, which try to identify potential targets of *a given pointer*, *storeless* models are designed to compute abstract relationships between heap-directed pointers themselves.

The goal of a storeless heap analysis is to statically estimate the *structure of the heap at each program point*. A typical heap analysis should be able to answer the question: "can two heap references at a given *program point*, lead to the same heap location?" This question can also be rephrased as: "are the two references aliased at the given program point?" For example in Figure 1.2(a), at program point `S`, the heap references `p->item` and `q->item` would access the same heap location, while the references `p->item` and `r->item` would not. Note that in the light of theoretical results establishing the undecidability of calculating precise alias relationships via static analysis [Lan92a], the analysis is not expected to give precise information. It is allowed to err conservatively i.e. two heap references may be reported to access the same location, even when they would not, in any execution of the program.

1.2 Pointer Analysis – A Historical Perspective

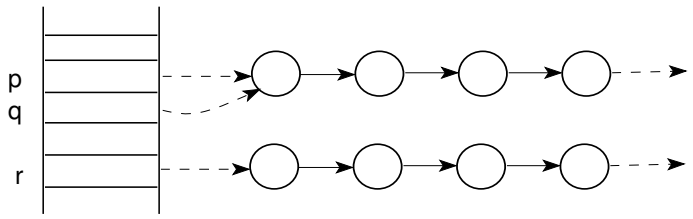
In this section, we summarize the large body of work done on the pointer analysis problem. We first discuss the recently proposed approaches that focus on the stack-directed pointer problem and only conservatively estimate the heap. We then present the techniques mainly designed for an accurate heap analysis.

1.2.1 Stack Pointer Analysis

The first alias analysis methods focused on calculating the aliases generated by the association of actual arguments with call-by-reference parameters. As mentioned

```
{   p = build_list();
    r = build_list();
    q = p;
S:  p->item = q->item + r->item;
    p = p->next;
    r = r->next;
T:  p->item = q->item + r->item;
}
```

(a) Program



(b) Heap Structure

Figure 1.2: Example Program for Heap Analysis

before, this aliasing can be considered as a restricted case of stack-based pointer aliasing. This sort of aliasing is characteristic of languages like FORTRAN and has the following important properties: (i) only call-sites contribute to the creation of aliases, and (ii) an alias relation valid at the entry of a procedure holds throughout its body. Effective techniques for computing such aliases have been developed and described [Bar78, Ban79, Mye81, Coo85, CK89]. Most of these analyses collect the alias information in two passes over the program: (i) an introductory pass to compute all the obvious aliases generated at call-sites, (ii) a propagation pass to propagate the aliases through the call-graph of the program.

Early work on pointer aliasing includes: (i) a flow-insensitive algorithm from Weihl [Wei80] which turned out to be very imprecise in practice [Lan92b], (ii) an approach proposed by Chow and Rudmik [CR82] for their CHILL compiling system, which handles only single-level pointers, and (iii) a study by Coutant [Cou86] of Weihl's algorithm, which suggested the use of pragmas for collecting more accurate information.

Landi and Ryder [LR92] designed and implemented one of the first flow-sensitive interprocedural alias analysis algorithms for C programs. They collect alias information in the form of pairs of *object names*. An *object name* consists of a variable and a (possibly empty) sequence of dereferences and field accesses. Typical alias pairs are: `(**a, *b), ( *(a->next), *(b->next) )`. In the presence of recursive data structures, the number of object names is infinite. To avoid this, they k-limit object names, where no object name can have more than `k` dereferences. For example, for `k = 1`, `p->l->r` would be represented by `p->l`. They also name heap objects according to the malloc site that allocates them.

Their method is precise in the presence of single-level pointers, and effectively resolves stack-based aliases in the presence of multiple-level pointers. It is not designed to accurately handle heap-allocated recursive data structures. In some special cases it can help detect completely unaliased data structures (lists and trees) built by a program, but neither empirical nor theoretical evidence is available to draw any general conclusions.

Choi, Burke and Carini [CBC93] presented another algorithm for flow-sensitive alias analysis. They compute aliases as pairs of *access paths*. Their access paths are similar to object names [LR92]. However, they do not use access paths to name heap objects. They use the place (statement) in the program, where an anonymous

heap object is created, to name it. To avoid giving the same name to heap objects created at the same statement, but along different call-chains, they qualify the names with procedure strings. In the presence of recursion, this qualification proves to be of limited use. They mention that they combine this naming scheme with k-limiting to analyze recursive structures. It is not clear from their paper, what type of k-limiting they perform [MLR+93]. They also propose the idea of transitive reduction of alias pairs into points-to pairs.

Emami, Ghiya and Hendren [EGH94] proposed the approach of decoupling stack and heap analyses. They focus on analysis of stack-directed pointers, and collect alias information in the form of points-to relationships. A points-to relationship is denoted by a triple `(p, x, d)`, which indicates that pointer `p` definitely (`d = D`) or possibly (`d = P`) points-to the location named `x`. As locations in the heap are anonymous, they are represented by one abstract location called *heap*. All heap-directed pointers are reported to be pointing to this location.

The points-to abstraction provides a more compact representation for calculating aliases than exhaustive alias pairs based on access paths. It also enables simultaneous calculation of both possible and definite relationships. Empirical results reported indicate that this method collects highly precise information for stack-based aliases. At the same time, it builds a framework for conducting a variety of heap analyses (discussed in sections 2.4 and 2.6).

Wilson and Lam [WL95] also use a points-to abstraction. Additionally they use the concept of *location sets* to name both a block of memory (arrays and structures), and a set of positions within that block. Other approaches discussed above only use one name to represent aggregate structures like arrays and structures. They name heap objects in the same fashion as [CBC93]: allocation site qualified with a procedure string.

All the techniques presented above handle procedure calls in a context-sensitive manner i.e. the effect of a procedure call is estimated specific to a calling context, and not just summarized for all possible calling contexts. They use different strategies to abstract calling contexts: assumed alias sets [LR92], last call site and source alias sets [CBC93], invocation graphs [EGH94], and partial transfer functions [WL95]. In addition, Emami et. al [EGH94] precisely handle indirect calls through function pointers in C. Wilson and Lam follow their strategy for handling function pointers.

Ruf [Ruf95] compared a context-insensitive alias analysis with a context-sensitive

analysis. He reported that for his benchmark suite the latter provides more accurate information in general. However, this additional accuracy is not seen at indirect references where the analysis information is actually used. Our belief is that while this may be true for programs with a simple interprocedural structure, for larger programs with more complex call-graphs context-sensitivity is required.

Steensgaard [Ste96] proposed a type-inference-based near linear time (in the size of the program) points-to analysis algorithm, which is both flow- and context-insensitive. Types are used to represent locations of variables as well as possible runtime contents of these locations. One type is assumed to be pointing to another type, if any program variable represented by the former, can point to a variable represented by the latter. Two types are merged (unified) during analysis, when a potential assignment is seen between program variables represented by them.

Shapiro and Horwitz [SH97b] compare Steensgaard's algorithm with that of Andersen [And94]. In contrast to Steensgaard, Andersen's algorithm represents each program variable with one type variable, and allows one type variable to point to several type variables. It also has cubic time complexity. Their study reports that Andersen's algorithm provides more accurate information for majority of the programs, however with increased cost. They also parameterized Andersen's algorithm in the number of program variables that can be represented by a type variable, and the number of points-to out-edges allowed for a type variable, and reduced its cost to within a factor of log N of Steensgaard's algorithm.

The above two approaches are promising for analyzing industry-scale large programs. They can provide accurate information for programs with simple call-structure. For more complex programs, the information provided by them can be used to reduce the search space of more precise pointer analyses. Such an approach is proposed in [ZRL96]. However, these approaches also provide a very conservative approximation for heap, based on malloc sites.

1.2.2 Heap Pointer Analysis

Much of the earlier work on pointer analysis, was devoted to heap analysis, more specifically to analysis of heap-based recursive data structures. This was because these analyses were designed for languages like LISP and Scheme which do not support the address-of (`&`) operator like C, and hence do not have stack-directed pointers.

Jones and Muchnick [JM81] proposed one of the first approaches to the heap analysis problem. They analyze LISP-like structures for a simple language without procedures. They abstract the structure of the heap at each program-point, in the form of a set of graphs. Nodes in the graph represent objects in the heap, while edges represent the links between these objects. Nodes bound to variables are labeled by variable names. Nodes which can possibly be shared (have more than one parent) or become part of a cycle, are respectively labeled as shared and cyclic. They use set union as data flow merge operator at join points, which results in a set of graphs at each program point. Since the graphs abstract recursive structures, they can have unbounded number of nodes. To avoid building infinite graphs they use the notion of *k-limiting*, whereby all the nodes in a graph accessible from a variable after traversing k or more links, are coalesced into one summary node. For example, a 2-limited linked list is shown in Figure 1.3.

The goal of their analysis is to optimize storage allocation. The k-limited graphs at all program points are analyzed to classify variables into three categories: (i) variables which cannot access any shared or cyclic nodes at any program point, (ii) variables which can access shared nodes but no cyclic nodes, and (iii) variables which can access cyclic nodes. The heap cells accessible from the first variety of variables can be deallocated as soon as pointers to them are destroyed, as their reference count never exceeds one. Those accessible from the second variety can be reference-counted while the rest need to be garbage-collected.

Although an interesting analysis for improving storage allocation, it does not provide precise enough estimation of the heap structure for program optimization. Due to k-limiting all the information about nodes beyond depth k is lost. The introduction of summary nodes can generate spurious cycles in otherwise acyclic structures. Finally, maintaining a set of graphs at every program point, can prove to be quite expensive.

Jones and Muchnick [JM82] also proposed a flexible framework for analysis of programs with recursive data structures. They designate program points which create or modify recursive structures with tokens. The tokens can be considered as local representations of the data structures at the given program points. They then define a retrieval function to finitely represent the relationships between tokens and data values. The definition of the retrieval function is based on the simulation of program statements, using abstract interpretation [CC77]. The analysis framework is parameterized by the choice of token sets. Thus a wide range of analyses can be expressed

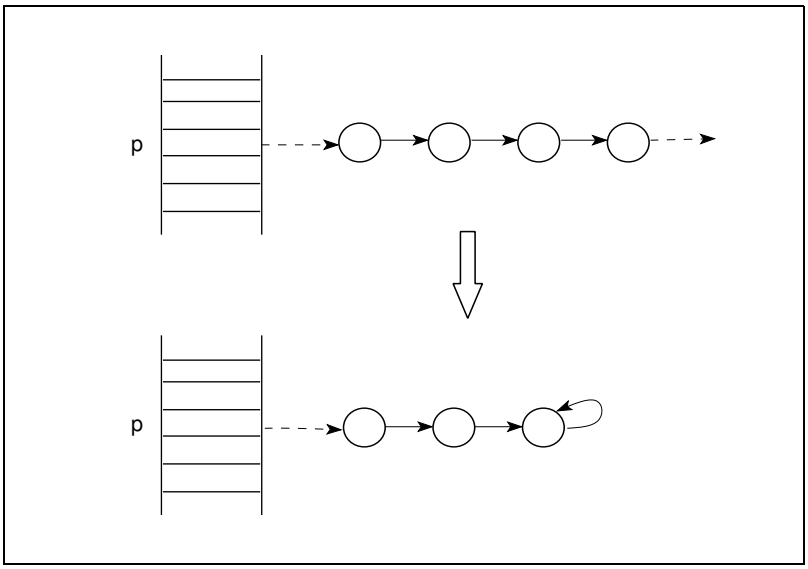


Figure 1.3: An Example of 2-limiting

in this framework. However, this method has remained mostly of theoretical interest, being expensive in both space and time.

Larus and Hilfinger [LH88] use a variation of k-limited graphs called *alias graphs* for analyzing Lisp programs. Their goal is to detect potential conflicts between heap accesses at different program statements. They label edges in the alias graph by names of corresponding accessors (pointer fields). In addition, they label nodes either by a variable name or by a path expression. Path expressions are regular expressions summarizing possible access paths from a variable to the node being labeled. A newly allocated node is labeled by an aggregate of the labels of the arguments to the allocation function(cons). This proves to be more precise than labeling the node by the program point where it is allocated. Unlike [JM81], they maintain only one alias graph per program point instead of a set of graphs. They define a meet operator that combines two alias graphs into a new alias graph that contains all aliases in either graph. To keep the size of the resulting graph finite, they introduce summary nodes using s-l limiting. In an s-l limited alias graph, no node has more than s outgoing arcs (except the node representing the bottom element), and no node has a label longer than l.

Once an alias graph is computed for each program point, conflict detection is

done. A potential conflict exists if access paths at the given statements can lead to a node with the same label in their respective alias graphs. This method works well only for simple data structures like trees and lists. It is rendered expensive by its complex meet, node summary and node labeling operations.

Horwitz, Pfeiffer and Reps [HPR89] presented another variation on k-limited graphs, called storage graphs which abstract the dynamic store. They present a variety of ways to k-limit the storage graphs. The goal of their analysis is detection of dependences between program statements. To this end, they label each node in the storage graph with the program point that last set its contents (unlike Larus and Hilfinger, who use path expressions for labeling nodes). A statement `S` is (flow) dependent on statement `T`, if `S` reads a location whose abstraction in the storage graph is labeled with statement `T`.

They maintain a set of storage graphs at each program point, unlike a single alias graph [LH88]. This makes their analysis more precise, but also more expensive. They use abstract interpretation [CC77] augmented with an instrumented semantics, to prove the correctness of their technique. However, it is unclear how effective it would prove to be in practice.

Another approach to abstracting the heap structure in the form of a bounded graph was given by Chase, Wegman and Zadeck [CWZ90]. Their abstraction, called storage shape graph (SSG), contains one node for each variable and one for each allocation site in the program. It is based on the premise that nodes allocated at different places tend to be treated differently, while the ones allocated at a given site would be updated similarly. This abstraction can introduce cycles in otherwise unaliased structures like lists and trees. For example, if all nodes of a list are allocated at the same site, they would be represented by a single summary SSG node with a self-cycle.

To avoid this possibility, they augment their abstraction with reference counts for each node, where nodes with reference counts less than two, would represent trees and lists. Further refinements to the model include: keeping multiple *interesting* instances of an allocation site (i.e. SSG node), enabling *strong updates*, and defining a precise meet operator for join points. An SSG node is considered interesting if it is pointed to by a *deterministic* variable i.e. a variable which does not point to any other node except possibly to `nil` node. A strong update involves replacing edges leaving a node with a new set of edges, giving more precise information. It can only

be performed for nodes representing a single heap location. Finally, the meet operator tries to minimize the creation of summary nodes, and only merges nodes representing the same allocation site.

This method would give precise results in some special cases. In general, it can be overly conservative because of one SSG node abstracting several run-time locations. For example, it would give highly imprecise results, if the program uses a single routine for allocating nodes (authors suggest the use of function inlining to overcome this.). Further, the meet operation is fairly complex.

Plevyak, Chien and Karamcheti [PCK94] have extended the model of Chase et al. [CWZ90] to handle regular cyclic structures like doubly linked lists and trees with parent pointers, more precisely. They introduce additional nodes called choice nodes, to represent that two given links coming into a summary node would not exist at the same time. They also annotate summary nodes with identity paths, to indicate which combinations of link fields can create cycles. Presently they do not handle procedure calls. Further, their analysis needs empirical verification, though they give some examples in the paper.

Recently, Sagiv et. al [SRW96] have presented a new approach to estimate the *shape* properties of heap data structures. They also abstract the heap as a storage shape graph [CWZ90]. However, their graph contains nodes only for heap locations pointed to by program variables (i.e. stack-resident heap-directed pointers). The rest of the heap locations are represented by one summary node. Each non-summary node is labeled by a name set, which includes all program variables that *must* point to this node at the given program point, considering all possible execution paths. The *must* property of the name sets enables them to perform strong updates for statements involving destructive updates. However, it can also lead to an exponential number of name sets, specially in the presence of loops and recursion. Their approach also involves a rather elaborate process of *materializing* new shape graph nodes from existing ones. This is required for statements of the form `x = y->next`, which give a name to an existing (hitherto anonymous) heap node and hence needs to be incorporated in the shape graph.

The heap analysis techniques described so far are *store-based*, as they attempt to explicitly abstract the dynamic store in the form of a bounded graph. They basically differ from each other in the way they choose to bound the graph. Further, nodes in the graph are sometimes labeled to facilitate conflict detection between

statements [LH88, HPR89]. Procedure calls are either not handled [JM81, HPR89, PCK94, SRW96] or are analyzed with different degrees of precision [JM82, LH88, CWZ90]. The restriction of representing several heap locations with one abstract location, forms the main source of imprecision for the *store-based* techniques.

To avoid this trap, Hendren and Nicolau [HN90] took a different approach. They focus on abstracting the properties of data structures being built and manipulated, instead of abstracting each cell in the heap. Their main focus is on identifying data structures with regular properties like trees and DAGs. The knowledge about the underlying data structures is then used for interference analysis and parallelization. For example, computations on left and right subpieces of a binary tree can be scheduled in parallel.

To collect such information, they perform *path matrix analysis*. A path matrix `P`, is a matrix of *stack-resident* heap-directed pointers, called *handles*. An entry `P[r,s]` in the matrix contains the summary of possible access paths from pointer `r` to pointer `s` in the heap, at the given program point. Access paths consist of link fields, and are represented as restricted regular expressions called *path expressions*. Path relationships between pointers are used to determine when a tree temporarily becomes a DAG. DAG nodes are reference-counted to detect when they again become tree-like (i.e. when the reference count becomes less than two). They perform context-sensitive interprocedural analysis and handle recursion precisely, which is important, as recursion is the main tool to build and use recursive data structures. Once the path matrix analysis determines the underlying data structure to be a tree, they perform interference analysis based on this information. This exposes the divide and conquer type of parallelism, induced by recursive traversal of tree-like structures.

Their method is precise and effective for trees and to some extent for DAGs. However, it cannot handle cyclic structures, which are commonly used in programs, like doubly-linked lists and trees with parent pointers.

Deutsch [Deu92, Deu94] calculates aliases in the form of pairs of *symbolic access paths*. This abstraction is particularly suited to recursive data structure analysis. A symbolic access path (SAP) is an access path possibly containing symbolic expressions of the form B^k, where B is a set of access paths called a *basis* and k is a variable. For example, the SAP `X->`$(tl)^i$`->hd`, has its *basis* as tl. This SAP, when parameterized on i, finitely represents an infinite number of access paths from the head of a list to the `hd` fields of its nodes. No imprecision is incurred, as happens with the k-limiting of object names [LR92].

An alias pair in this framework consists of a pair of symbolic access paths qualified by an equation. Thus a *position dependent* alias relationship of the form: "the ith element of list X is aliased to the $2i + 1$th element of list Y", would be precisely expressed as `(< X->`$(tl)^i$`, Y->`$(tl)^j$` >, j=2i+1)`. Although a more powerful and expressive framework, it is not clear if it is practical enough to be implemented in a real compiler.

Besides the automatic analysis techniques discussed above, certain language-based approaches have been proposed to get the information from the programmer. A brief discussion follows.

Lucassen and Gifford [LG88] defined a language (FX-87), which incorporates both an *effect* and a *type* system. The effect of a computation must be explicitly associated with a region of memory. The effect system differentiates between totally disjoint linked structures, but fails to distinguish between disjoint subpieces of a data structure.

Hendren, Hummel and Nicolau [HHN92a, HHN92b] presented a mechanism called ADDS (Abstract Description of Data Structures), to explicitly convey the *dimension* and *direction* properties of a data structure, to the compiler. This involves enriching the type definitions of data structures with some semantic information. For example, consider an ADDS type definition of a doubly linked list:

```
type TwoWayLL [X]
 { int       data;
   TwoWayLL *next is uniquely forward along X;
   TwoWayLL *prev is uniquely backward along X;
 };
```

This description tells the compiler that if the list is traversed using only `next` links, then all the nodes visited are unaliased. They demonstrate that using such information, the compiler can perform useful transformations like software pipelining.

Hummel, Hendren and Nicolau [HHN94] presented a more formal approach, to convey the alias properties of data structures. They propose a language based on regular expressions, which captures alias properties in terms of axioms applicable to the type definition of a data structure.

With their approach, the alias properties of a doubly linked list can be easily expressed as:

$\forall$ p, p.next <> p.ϵ
$\forall$ p, p.prev <> p.ϵ
$\forall$ p, p.next.prev = p.ϵ

Using these aliasing axioms, fairly complicated data structures like sparse matrices and two-dimensional range trees, can be precisely described. The ADDS descriptions can also be translated to aliasing axioms.

They also present a general purpose dependence test for dynamic data structures. To detect dependence between two program statements, they first traverse the program segment to find the relative position of the heap locations accessed by them. This is achieved by determining the possible access paths to these heap locations, with reference to a pointer pointing to a fixed heap location. A theorem prover is then used to determine if the given access paths can lead to a common heap location, on the basis of the aliasing axioms provided for the data structure being traversed.

This technique is general purpose (i.e. is not restricted to data structures of certain types like lists and trees). The initial results provided in the paper are encouraging. It would be interesting to see more detailed experimental results.

Klarlund and Schwartzbach [KS93] also proposed a similar approach called *Graph Types* to describe data structures using regular-like expressions. With graph types, pointer fields are separated into two types, *tree* and *routing* fields. The tree fields must create a spanning tree for the data structure, and the routing fields are defined in terms of the tree fields and the underlying spanning tree. Thus, graph types can only describe data structures with a spanning tree backbone.

A large body of work on analysis of heap-allocated objects, has focused on other problems like reference counting and memory lifetimes [Hud86, ISY88, RM88, Deu90, WH92].

1.3 Pointer Analysis in the McCAT C Compiler

All the techniques we have discussed so far, either focus on stack analysis with a conservative approximation of the heap, or present sophisticated heap analyses considered in isolation from stack analysis (assuming stack-directed pointers do not exist).

The work reported in this thesis is built upon the pointer analysis framework implemented in the McCAT C compiler [HDE+93]. This framework decouples stack and heap analyses, instead of solving the two problems using the same abstraction. First points-to analysis [EGH94] is used for accurately estimating stack pointer relationships, approximating all heap locations as a single symbolic location *heap*. On the subset of pointers reported to be pointing to *heap*, two *storeless* heap analyses are applied: *connection* analysis [GH96a] and *shape* analysis [GH96b].

These heap analyses use simple *storeless* abstractions that capture boolean relationships between *stack-resident* heap-directed pointers in the program, computed at each program point. Below, we briefly describe the two analyses and identify their specific application domains:

- **Connection Analysis [GH96a]:** This analysis determines if two heap-directed pointers point to the same linked structure (i.e. they are connected) or to *disjoint regions* in the heap (i.e. they are not connected). It uses a *connection matrix abstraction*, which is a boolean matrix of heap-directed pointers, to collect *connection* information. This information is useful in disambiguating heap accesses to completely disjoint data structures like dynamically-allocated arrays and other non-recursively defined structures. Scientific applications written in C typically use these constructs.

- **Shape Analysis [GH96b]:** This analysis focuses on estimating the shape of the structure *accessible* from a given pointer: is it tree-like, DAG-like or a general graph containing cycles. It uses three simple abstractions to achieve this goal, which include:

  1. *Direction Matrix:* This abstraction approximates the *path existence* relationship between heap-directed pointers, i.e. if there exists an access path in the heap from one pointer to another pointer.
  2. *Interference Matrix:* This matrix computes, if a common heap location can be accessed, starting from two given heap-directed pointers. It is computed in conjunction with the direction matrix, and is designed to handle DAG-like structures, where two pointers may not have a path to each other, but can still *interfere* i.e. access a common heap location. It forms a superset of the direction matrix.

3. *Shape Attribute:* This attribute is associated with each heap-directed pointer to store the shape of the data structure accessible from the given pointer.

The motivation behind shape analysis is to identify tree and list-like structures in programs in a simple and efficient way. This knowledge can then be gainfully exploited for parallelizing programs [Lar89, Hen90], and performing optimizing transformations like loop unrolling [HG92] and software pipelining [HHN92a]. There is a large body of applications which use trees and lists as principal data structures.

It can be noted that these abstractions are practical variations on the path matrix model of Hendren and Nicolau [HN90]. The differences lie in collecting coarser path information for efficiency reasons, and associating additional attributes with each pointer (e.g. shape attribute).

In the McCAT C compiler, these pointer analyses are applied in a *hierarchical* fashion. If the points-to analysis reports no heap-directed pointers, no heap analysis needs to be performed. Otherwise, a simple and cheap analysis like connection analysis is applied. If it provides overly conservative results, then the more advanced shape analysis is used. Thus, the cost of an expensive analysis is incurred only if cheaper analyses do not provide accurate information for the given program. Further, information obtained from analyses earlier in the hierarchy, is used to make further analyses more efficient.

1.4 Using Pointer Analysis

Our discussion so far has focused on pointer analysis itself. We now shift attention to the problem of how to use pointer analysis for various compiler applications, *which is the focus of this thesis.* We first illustrate how the presence of pointers affects compiler optimizations.

1.4.1 Pointers and Compiler Optimizations

Compiler optimizations strive to make existing programs run faster, typically by eliminating redundant computations, reducing memory traffic, exposing parallelism and

hiding memory latency. The fundamental requirement for any compiler optimization is accurate identification of the set of locations *read/written* by a given program statement or a program region (read/write set computation). The goal of pointer analysis information is to enable accurate computation of read/write sets, when statements involve pointer indirections. The main idea is to expose the set of memory locations accessed via an indirect reference.

Without pointer analysis information, compilers need to make very conservative assumptions about the set of memory locations accessed by an indirect reference. This leads to very imprecise read/write set information, which in turn considerably inhibits the power of compiler optimizations. We illustrate this below using the example program fragments in Figure 1.4.

Consider the program fragment in Figure 1.4(a). The loop body contains several redundant computations. First, the expression `x * i` need not be recomputed for statment `V`, as it is already computed at statement `S`, and the values of `x` and `i` are not changed between `S` and `V`. Second, the expression `x * y` in statement `T` is invariant for the loop, as both `x` and `y` are not updated inside the loop, and need not be computed for every iteration. An optimizing compiler is normally able to detect and eliminate such redundant computations via optimizations known as *common subexpression elimination* (CSE) and *loop-invariant removal* (LIR). The CSE optimization would store the value of `x * i` into a temporary, say `tx`, at `S` and then substitute `x * i` at `V` with `tx`. Similarly, the LIR optimization would store the value of `x * y` into a temporary, say `txy`, before the loop, and then use `txy` in statement `T`. The optimized version of the program fragment is shown in Figure 1.4(b).

However, without pointer analysis information, these optimizations cannot be applied to our example program fragment. The main problem is posed by pointer indirections at statement `U`. In the absence of pointer analysis information, the compiler has to conservatively assume that the indirect reference `*pz` on *lhs* of statement `U` *can write to any memory location.* Thus CSE cannot be applied to the expression `x * i`, as statement `U` can potentially update both `x` and `i` between `S` and `V`. For similar reasons, the expression `x * y` cannot be considered loop-invariant.

Another side-effect of the above conservative assumption impacts register allocation. Register allocation [Bri92] is an important program optimization that assigns frequently used variables to registers, instead of loading/storing their values from memory on each access. This is significant because for modern microprocessors, register accesses are an order of magnitude faster than memory accesses. With this

```
px = &x; py = &y; pz = &z;
init_values(px, py);
sumX = 0; prod = 0;
for (i = 0; i <= N; i = i + 1)
{
  S: sumX = sumX + x * i;
  T: prod = prod + x * y;
  U: *pz = *pz + sumX + prod;
  V: sumT = sumT + x * i +
                   y * i;
}
```

(a)

```
px = &x; py = &y; pz = &z;
init_values(px, py);
sumX = 0; prod = 0;
txy = x * y; /* LIR */
for (i = 0; i <= N; i = i + 1)
{
     tx = x * i; /* CSE */
  S: sumX = sumX + tx;
  T: prod = prod + txy;
  U: *pz = *pz + sumX + prod;
  V: sumT = sumT + tx + y * i;
}
```

(b)

```
p = build_list();
q = build_list();
while (p != NULL)
{
  M: p->i = p->i + q->i;
  N: p = p->next;
}
```

(c)

```
p = build_list();
q = build_list();
tq = q->i; /* LIR */
while (p != NULL)
{
  M: p->i = p->i + tq;
  N: p = p->next;
}
```

(d)

Figure 1.4: Motivating Examples

knowledge, the compiler should assign all variables used inside the loop to registers. However, it cannot do so due to the potential *back-door* access problem [HS90]. This problem arises when an access to a variable assigned to a register, can go to its memory location instead. We illustrate it below.

Suppose the compiler assigns variable `x` to register `regX`. Subsequently, the read references to `x` at `S`, `T` and `V` will read the value from `regX` instead of the memory location `x`. Now as per the worst-case assumption, the *lhs* indirect reference `*pz` at `U` can potentially write to the memory location `x`. However, such a write to `x` will actually go to the memory location of `x` and not to `regX`. Subsequently statements accessing `regX` will read the wrong value of `x`, and the optimization will violate program semantics.

To alleviate the impact of pointer aliasing, some compilers restrict the worst-case assumptions to *address-exposed* variables, i.e., an indirect reference is assumed to potentially read/write any address-exposed variable, as opposed to any variable. A variable is considered address-exposed if its address has been taken somewhere in the program. With this restriction, `*pz` at `U` can still potentially write variables `x`, `y` and `z` but not `sumX`, `prod` and `sumT`. Thus the latter three variables can now be register-allocated.

Finally, if the compiler uses the points-to analysis, it would have the points-to triple `(pz, z, D)` valid at statement `U`, indicating that pointer `pz` definitely points to the variable `z`. From this information, it can infer that the statement `U` only writes to the location `z`. Thus the compiler can apply the CSE and LIR optimizations as shown in Figure 1.4(b), and also promote all variables inside the loop to registers.

Besides alleviating the impact of pointer aliasing on traditional optimizations, pointer analysis information can also enable them to be applied to pointer references themselves. Consider the code fragment shown in Figure 1.4(c). The heap pointer reference `q->i` at statement `M` is loop-invariant, and can be substituted with a temporary (`tq`), as shown in Figure 1.4(d). Now the scalar temporary `tq` can be register-allocated, as opposed to the heap reference `q->i`. This saves one memory access per iteration.

However, to detect `q->i` as loop invariant the compiler needs to make two checks: (i) the pointer variable `q` is itself loop-invariant, and (ii) the heap reference `p->i` does not conflict with the reference `q->i` in any loop iteration. Clearly these checks cannot be made without appropriate pointer analysis information. For example, we

can use the points-to information to check that variable `q` itself is not written inside the loop via any indirect reference(i.e. is loop-invariant). Further, using connection analysis information, we can determine that pointers `p` and `q` point to disjoint data structures (two disjoint lists), and hence the heap references `p->i` and `q->i` cannot conflict. Note that points-to analysis information would not suffice in this case. It would report both `p` and `q` to be pointing to the symbolic location *heap*, thereby conservatively indicating a potential conflict.

Above, we have shown the interaction of pointer analysis with scalar compiler optimizations. Pointer analysis information is also crucial for more advanced compiler applications like array dependence testing, identifying coarse-grain parallelism, and for program understanding. We will discuss the usefulness of pointer analysis information for these applications in detail in chapter 4.

1.4.2 Computing Read/Write Sets

In the previous section, we illustrated the importance of pointer analysis information for compiler optimizations. We also demonstrated that the main goal of pointer analyses is to enable accurate computation of read/write sets for statements involving pointer indirections. In this section, we briefly sketch the computation of read/write sets from pointer analysis information.

Let us compute read/write sets for the code fragment shown in Figure 1.5. It is straightforward to use points-to information to compute read/write sets. For example in Figure 1.5, inside function `sum` we have the points-to information {`(c, pointA, D), (d, pointB, D)`}, indicating that pointers `c` and `d` definitely point to structures `pointA` and `pointB` respectively. Based on this information, one can determine that the indirect reference `c->x` on the *lhs* of statement `U` actually involves a *write access* to the stack location `pointA.x`, and the indirect reference `d->y` on the *rhs* involves a *read access* to the stack location `pointB.y`. Resolving all indirect references in a similar fashion, we get the following read/write sets for the function `sum`:

Read(U) = {c, d, pointA.x, pointB.y} Write(U) = {pointA.x}
Read(V) = {c, d, pointA.y, pointB.x} Write(U) = {pointA.y}
Read(sum) = {c, d, pointA.x, pointA.y, pointB.y, pointB.x}
Write(sum) = {pointA.x, pointA.y}

From this information one can easily detect that there are no interstatement dependencies inside function `sum`. Now consider function `flip`. Here both pointers `a` and `b` are heap-directed. The points-to information {`(a, heap, P), (b, heap, P)`} gives very conservative read/write sets: Read(S) = Read(T) = {a, b, *heap*}, and Write(S) = Write(T) = {*heap*}. These sets indicate a flow dependence from `S` to `T` i.e. statement `T` reads a location written by statement `S`, which is clearly a false dependence.

To get sharper read/write sets for function `flip`, we need to use heap analysis information. Let us use connection analysis. Connection analysis would report that pointers `a` and `b` point to disjoint heap objects, at the entry of function `flip`, and also at statements `S` and `T` inside the function. However, it does not provide us the information if pointer `a` *at statement* `S` points to a disjoint heap object from pointer `b` *at statement* `T`. This is the information required to prove that there is no flow dependence from statement `S` to statement `T`.

```
typedef struct point
{ double x;
  double y;
  struct point *next;
}Point;

Point pointA, pointB;
```

```
void main()
{ Point *s, *t;
    s = &pointA;
    t = &pointB;
 C: sum(s, t);
    s = alloc_point();
    t = alloc_point();
 D: flip(s, t);
}
```

```
void flip(Point *a, Point *b)
/* a gets the reflection of b */
{ Point *tmp;
  S: a->x = b->y;
  T: a->y = b->x;
}
```

```
void sum(Point *c, Point *d)
{ U: c->x = c->x + d->y;
  V: c->y = c->y + d->x;
}
```

Figure 1.5: Example for Computing Read/Write Sets

The main problem is that unlike points-to information, connection information does not enable us to resolve an indirect reference into a set of named memory locations. Connection analysis is a storeless analysis. It computes highly accurate

information at a given program point, but does not provide a mechanism to relate information across two distinct program points, or to summarize this information over a program region like a function body. To tackle this problem, we have introduced the notion of *anchor handles*. Anchor handles are symbolic names generated to capture heap relationships with respect to specific incarnation of pointers. We will discuss them in detail in section 3.3.

Finally, shape information is used to detect coarse-grain parallelism in programs with recursive data structures. In this case, points-to and/or connection information does not suffice, as it requires identification of disjoint sub-pieces of the same data structure. However, to use shape information we do not encode it explicitly into read/write sets. Here, we use special techniques that are based on collecting *access paths* to nodes in a data structure, and subsequently using shape information to determine if the given paths can lead to the same node. We discuss these techniques in detail in chapter 5.

1.5 Thesis Goals and Organization

Our main goal was to put the existing pointer analyses in the McCAT compiler framework to work, i.e., use the information provided by them for enabling more powerful program transformations. The main contributions of this thesis include:

Computing read/write sets based on a storeless pointer analysis: We have designed and implemented a connection analysis augmented with anchor handles, that computes heap read/write sets relative to these handles (chapter 3).

Applications based on read/write sets: Based on read/write sets, we demonstrate how to use the information for a wide variety of applications (chapter 4). These applications include:

1. Extending standard scalar compiler transformations, like loop-invariant removal, location-invariant removal, and common subexpression elimination, to include pointers references [GH98] (section 4.1);
2. Reducing communication overhead and exposing more parallelism for parallel EARTH-C [HTZ+96] programs (section 4.3);

3. Providing improved input to array dependence testers [GH98] (section 4.4);
4. Providing summary information that is useful for program understanding [GH98], dynamic compilation [APC+96] and prefetching of pointer data structures [LM96] (section 4.6).

Implementations and Empirical studies: We have implemented our techniques in the McCAT C compiler, and we present empirical data to illustrate the costs and benefits of the techniques (chapter 4). By performing source-to-source scalar transformations based on our read/write sets, we demonstrate performance improvement of 3.4% on average, and up to 10.3%, over the GNU C compiler [Sta92] compiling at the highest level of optimization. For array dependence testers we show significant improvements with pointer read/write sets, and we demonstrate the use of our read/write sets for program understanding via a tool that produces output that can be browsed via Web browsers. The benefits of our scalar optimizations on pointer references, have also been studied in the context of parallel EARTH-C programs, where they provide up to 25% performance improvement [TGHG97].

Detecting parallelism in programs with recursive data structures: We have developed effective techniques to use program-point specific shape information in combination with our read/write set information, to identify coarse-grain parallelism in C programs with recursive data structures (chapter 5). In particular, our techniques are tailored to identify function calls and loop iterations that access disjoint parts of tree- and list-like recursive data structures, and can be executed in parallel. We have implemented the techniques in the framework of the McCAT C compiler, and evaluated their effectiveness with respect to a set of C benchmark programs.

The rest of this thesis is organized as follows. In chapter 2, we describe the overall setting for the implementation of our pointer and read/write set analyses in the McCAT C compiler. Here, we also briefly describe how to compute points-to based stack read/write sets. In chapter 3, we specifically focus on computing connection-based heap read/write sets. In chapter 4, we illustrate scalar optimizations and discuss other applications of read/write sets. In chapter 5, we briefly describe the rules for estimating shape information and discuss how to use it for exploiting coarse-grain parallelism. In chapters 4 and 5, we also discuss relevant related work. Finally, in chapter 6, we draw conclusions and discuss the scope for future work.

Chapter 2

Setting

In this chapter, we outline the setting in which our analyses and transformations have been designed and implemented. We chose C as the language under analysis, and the McCAT C compiler as the platform for our implementations. The C language supports all the interesting and challenging pointer features and is widely used. The McCAT C compiler provides a simple and structured intermediate representation called SIMPLE, and a complete framework of pointer analyses.

We give a brief overview of the McCAT compiler in the first section, and of the SIMPLE intermediate representation in the next section. In section 2.3, we briefly describe points-to analysis and its interaction with heap analyses. We illustrate the interprocedural analysis framework in section 2.4. In section 2.5, we describe the computation of points-to based stack read/write sets. In the last section we briefly discuss the heap analyses and review the overall compilation framework, to put our goals in proper perspective.

2.1 The McCAT C Compiler

The McCAT C compiler is part of the McGill Compiler Architecture Testbed, being developed to study the interaction between smart compilation techniques and advanced architectural features [HDE+93]. The most important goal in its design was to develop appropriate intermediate representations to facilitate implementation of

various compiler analyses and transformations in a simple and straightforward manner. To this end, it supports an intermediate representation called SIMPLE, which is built from a high-level representation of the program, namely FIRST.

FIRST is a high-level Abstract Syntax Tree (AST) representation of the entire source program. It is built to separate the front-end processing (e.g. scanning, parsing, and type-checking) from the analysis, optimization and code-generation phases of the compiler. It retains the original format of the source program and its data structures. Analysis at this level can become cumbersome, specially if the program uses complex constructs and the programmer has resorted to various tricks allowed by high-level languages (specially C!).

In order to make the implementation of analyses simple and straightforward, FIRST is transformed to another AST intermediate representation called SIMPLE. As the name implies, SIMPLE breaks down all complex program constructs into a series of simple and regular constructs. It also makes control flow structured and explicit. SIMPLE forms the appropriate program representation for high-level analyses like alias and dependence analysis, and for high-level loop and parallelization transformations. We discuss it in more detail in the next section. A complete description is given in [Sri92].

The overall design of the McCAT compiler is shown in Figure 2.1. Note that the compiler takes as input a set of C files, which are linked by a source-level linker. This is necessary in order to have the whole source program available for interprocedural analysis. All the analyses and transformations are performed at the SIMPLE representation as shown in the figure. The transformed SIMPLE AST can either be directly given as input to the EARTH-C compiler [HTZ+96], which generates code for the EARTH-MANNA multithreaded architecture [HMT+95], or dumped back as an *optimized* C source file which can then be compiled with any native C compiler. Alternatively, the C-dump utility can generate HTML or C source files, annotated with the analysis information. The HTML files can be viewed via a Web browser for program understanding/debugging purposes. The annotated C files can be used as input by other research C compilers, which can retrieve the analysis information from the annotations, and subsequently use it for program optimization.

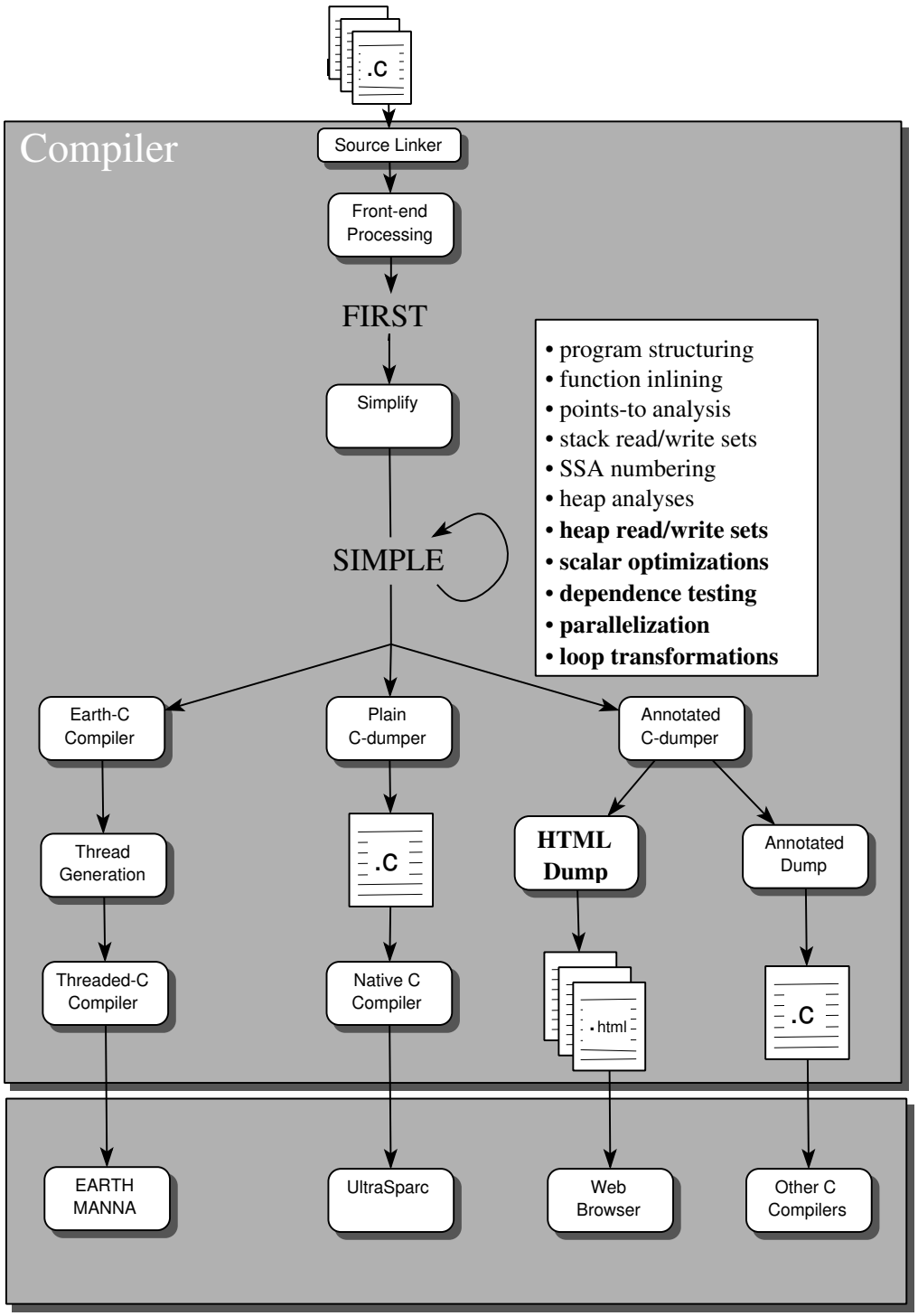


Figure 2.1: The McCAT Compiler

2.2 SIMPLE Intermediate Representation

The SIMPLE intermediate representation has been specially designed to facilitate accurate pointer analysis for C programs. Its major advantage lies in being simple to analyze. This is achieved by performing a number of simplifying transformations. Typical examples include: compiling complex statements into a series of basic statements, breaking down complex variable references into a series of simpler ones, simplifying procedure arguments to either constants or variable names, and moving variable initializations from declarations to statements in the body of the appropriate procedure.

SIMPLE restricts the number of basic statements in a program to fifteen. Different types of basic statements in a C program are broken down into one or more of these fifteen statements. In Figure 2.2 we list this set of basic statments. Note that variables '`x`' and '`y`' denote `varnames`, whereas the variables '`a`', '`b`', and '`c`' denote `vals`. The SIMPLE grammar for a `varname` is shown in Figure 2.3.

```
 1.   x  =  a binop b   where binop is any binary operation
 2.  *p  =  a binop b
 3.   x  =  unop a      where unop is any unary operation
 4.  *p  =  unop a
 5.   x  =  y
 6.  *p  =  y
 7.   x  =  f(args)     where args is a possibly empty list of arguments
 8.  *p  =  f(args)
 9.   x  =  (cast)b     where cast is any typecast
10.  *p  =  (cast)b
11.   x  =  &y
12.  *p  =  &y
13.   x  =  *q
14.  *p  =  *q
15.  f(args)
```

Figure 2.2: List of the 15 Basic SIMPLE Statements. Variables `x` and `y` denote `varname`. Variables `a`, `b`, and `c` denote `val`. Variables `p` and `q` denote `ID`.

Development of any new analysis is greatly simplified, as basic analysis rules need to be specified for only fifteen simple statements. Pointer analysis is further facilitated by the fact that only one level of indirection is allowed in any indirect variable

```
val : ID                          reflist : '[' val ']'
    | CONST                               | reflist '[' val ']'

varname : arrayref                idlist : idlist '.' ID
        | compref                        | ID
        | ID
                                  compref : '(' '*' ID ')' '.' idlist
arrayref : ID reflist                     | idlist
```

Figure 2.3: SIMPLE Grammar for a `varname`

reference. Indirect references of multiple level are broken down to adhere to this format during simplification. Figure 2.4 shows an example of simplifying indirect references. Indirect references augmented with field accesses (i.e. component references) like `(*a).next` and `(*a).next.item` , and indirect array references like `a[i]` where `a` is a pointer to an array, are represented in the basic statements (Figure 2.2) by variables '`x`' and '`y`' (as can be seen from the grammar shown in Figure 2.3). However, the level of indirection remains restricted to one.

```
**pp = q;        ⇒        temp1 = *pp;
                          *temp1 = q;
```

Figure 2.4: Simplification of an Indirect Reference

Another important feature of SIMPLE is that it provides a compositional representation of the program, and makes the control flow structured and explicit. The compositional control statement forms supported by SIMPLE are simplified versions of: statement sequences, `for`-loops, `while`-loops, `do`-loops, `switch/case` statements, and `if/else` statements. In addition, the `return` statement is supported for exiting a procedure, and `break` and `continue` statements are supported for exiting a loop. Since the unrestricted use of `goto` is not compositional, the compiler provides a structuring phase that eliminates all `goto` statements from a C program [Ero94, EH94].

With a compositional representation, structured analysis techniques can be used to analyze all control constructs. For example, a while loop can be analyzed by considering only its components: the conditional expression and the body. A structured analysis framework is easier to implement, as only one analysis rule needs to be defined for each of the compound statements such as conditionals and loops. Further,

it becomes easier to reason about the fixed-point computations for loop constructs. Our pointer analyses are performed at the SIMPLE level, using structured analysis techniques.

2.3 Points-to Analysis

In C, one can have pointers to locations on the stack as well as in the heap. As mentioned earlier, the McCAT pointer analysis framework decouples the analysis of stack-directed and heap-directed pointers. First points-to (alias) relationships of variables on the stack are resolved, using an analysis called *points-to* analysis. This analysis abstracts the set of all accessible stack locations with a finite set of named abstract stack locations. An *abstract stack location* may correspond to: (1) the name of a local variable, global variable or a parameter; or (2) a symbolic name that corresponds to locations indirectly accessible through a pointer dereference, when these locations correspond to variables not in the scope of the procedure under analysis; or (3) the symbolic name *heap* that represents all accessible heap locations. Note that symbolic abstract stack locations can represent more than one real stack location. We further elaborate on this in the next section.

Given that each real stack location has a corresponding named abstract stack location, pointer relationships are then captured in the form of definite and possible points-to relationships between abstract stack locations, defined as follows:

Definition 2.3.1 *Abstract stack location* `x` definitely points-to *abstract stack location* `y`*, at a given program point, if* `x` *and* `y` *each represent exactly one real stack location at that program point, and the real stack location corresponding to* `x` *contains the address of the real stack location corresponding to* `y`*. This is denoted by the triple* `(x,y,D)`*.*

Definition 2.3.2 *Abstract stack location* `x` possibly points-to *abstract stack location* `y`*, at a given program point, if it is possible that one of the real stack locations corresponding to* `x` *contains the address of one of the real stack locations corresponding to* `y` *at that program point. This is denoted by the triple* `(x,y,P)`*.*

The complete description of points-to analysis can be found in [Ema93] and an overview in [EGH94]. We demonstrate it on an example program, in Figure 2.5.

Part (a) of the figure shows the original program, while part (b) shows the simplified program decorated with program-point-specific points-to information. Note that all heap-directed pointers are reported to be possibly pointing to the abstract stack location *heap*. A pictorial representation of the abstract stack at program point D is shown in part (c) of the figure. Solid lines in the figure denote definite relationships while dashed ones represent possible relationships.

Points-to analysis lays the foundation for computing stack read/write sets and for performing heap analyses. It exposes the set of memory locations accessed via indirect references, which enables accurate computation of stack read/write sets. For example in Figure 2.5(b), using the points-to information `(pp, p, D)`, one can accurately identify that the statement `temp1 = *pp` in Figure 2.5(b), reads the stack locations `pp` as well as `p`. For heap analyses, points-to analysis determines the set of heap-directed pointers in the program. This set consists of pointers which are reported to be possibly pointing to *heap* at some point in the program. For example, for the simplified program in Figure 2.5(b), only pointers `p`, `q` and `temp1` would fall into this set. Heap analyses only need to approximate the relationships between these heap-directed pointers. This helps in reducing the storage requirements for the abstraction being implemented.

The points-to information is also used by heap analyses to accurately handle indirect references. For example, consider the analysis of statement `p->next = q` (`(*p).next = q`) in Figure 2.6. To estimate its effect on any heap analysis, we first need to know what locations pointers `p` and `q` can point-to. If both of them point-to *heap*, then this statement links the actual pointed-to locations through the `next` link, as shown in Figure 2.6(a). Similarly, if `q` points-to *heap*, and `p` points-to a stack-resident structure `x`, we would have `x.next` pointing to the same heap location as `q`, as shown in Figure 2.6(b). On the other hand, if `q` points-to a stack location, and `p` points-to the structure `x`, with its `next` field also pointing to a stack location, the statement does not affect heap analysis, as no pointer to a heap location is updated (Figure 2.6(c)).

2.4 Interprocedural Analysis Framework

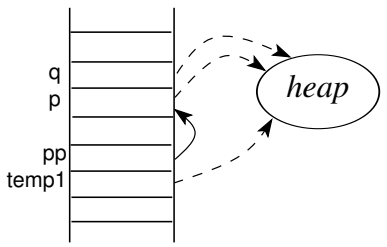
Points-to analysis handles procedure calls in a context-sensitive manner, i.e., it estimates the effect of a procedure call, within its specific calling context, and not as a

```
main()
{ int *p, *q, **pp;

  p = (int *) malloc();
  q = p;
  pp = &p;
  **pp = 5;
}
```

(a) Original Program

```
main()
{ int *p, *q, *temp1, **pp;

  p = (int *) malloc();
  /* A: { (p,heap,P) } */
  q = p;
  /* B: { (p,heap,P), (q,heap,P) } */
  pp = &p;
  /* C: { (p,heap,P), (q,heap,P)
        (pp,p,D) } */
  temp1 = *pp;
  /* D: { (p,heap,P), (q,heap,P)
        (pp,p,D) (temp1,heap,P) } */
  *temp1 = 5;
}
```

(b) Simplified Program with Points-to Information



(c) Abstract Stack Representation

Figure 2.5: An example for Points-to Analysis

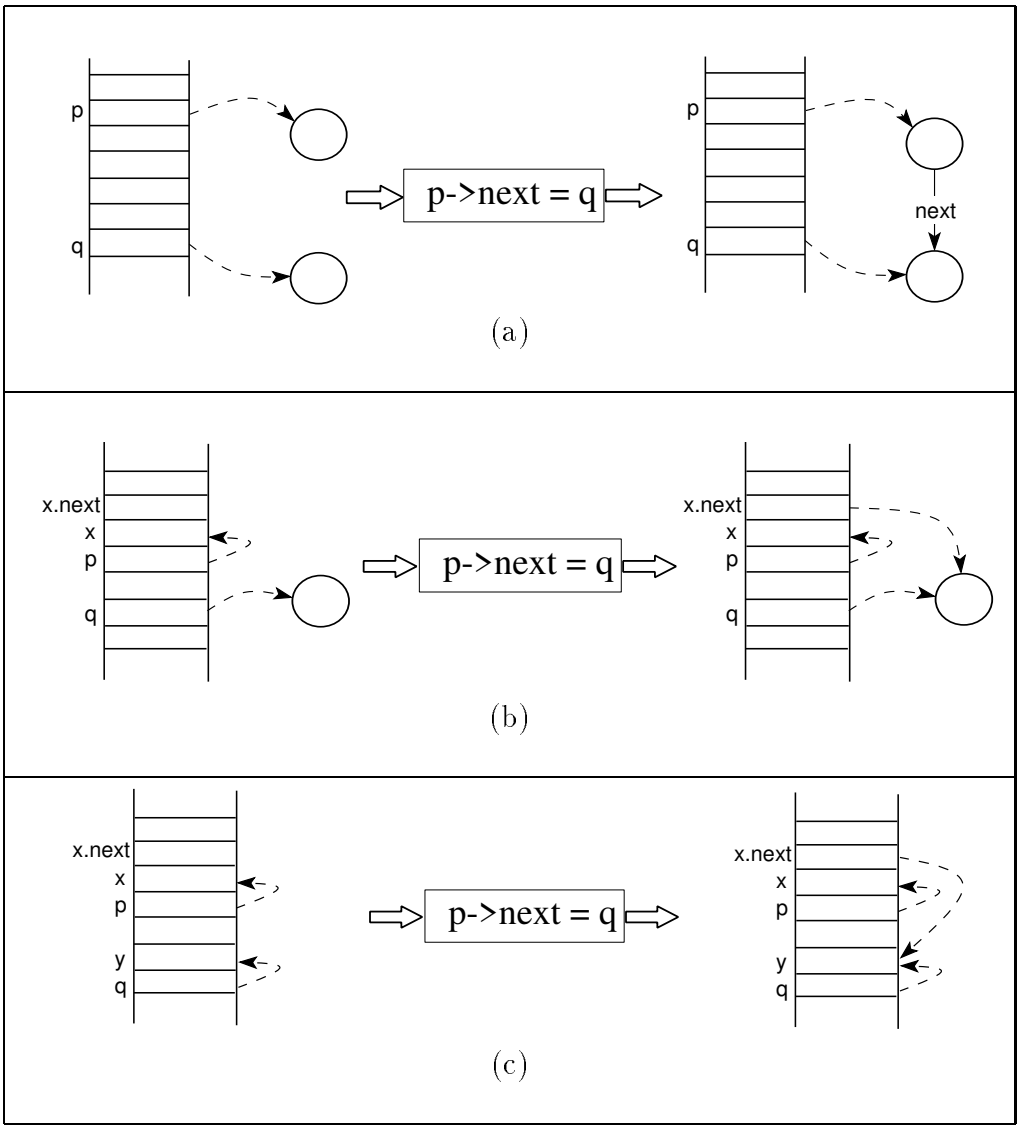


Figure 2.6: Points-to Information and Heap Analyses

summary of all possible calling contexts. To support this analysis strategy, it builds a framework for interprocedural analysis. Other context-sensitive interprocedural analyses like heap analyses, are built on top of this framework. A complete description of the framework can be found in [Ema93, HEGV93, EGH94]. Below, we briefly describe its salient features.

2.4.1 Representing Calling Contexts

In general, a calling context depends on the *invocation path* followed by the program, i.e., the chain of procedure invocations starting from `main` and ending with the procedure call under analysis. Points-to analysis builds an *invocation graph*, where all invocation paths are **explicitly** represented. In the absence of recursion, the invocation graph is constructed by a simple depth-first traversal of the invocation structure of the program. Consider for example, the invocation graph for the program in Figure 2.7(a). An important characteristic of the invocation graph is that each procedure invocation chain is represented by a unique path in it, and vice versa. Using the invocation graph one can distinguish not only calls from two different call-sites of a procedure (calls to f() in Figure 2.7(a)), but one can also distinguish two different invocations of a procedure from the same call-site when reached along different invocation chains (call to f() in Figure 2.7(b)).

In the presence of recursion the exact invocation structure of the program is not known statically, and one must approximate all possible unrollings of the recursion. Figure 2.8 illustrates a program with simple recursion and the set of all possible invocation unrollings for this program, and the invocation graph that is used to approximate all possible unrollings. To build the graph in the case of recursion one terminates the depth-first traversal each time a function name is the same as that of one of the ancestors on the call chain from `main`. The leaf node (representing the repeated function name) is labeled as an approximate node, and its matching ancestor node is labeled as a recursive node. The pairings of these nodes are indicated with a special back-edge from the approximate node to the recursive node. It should be noted that these back-edges are used only to match the approximate node with its appropriate recursive node, and they are therefore quite different from the other tree edges which correspond to procedure calls. This scheme is completely general. Consider, for example, the invocation graph for a program with mutual recursion displayed in Figure 2.9.

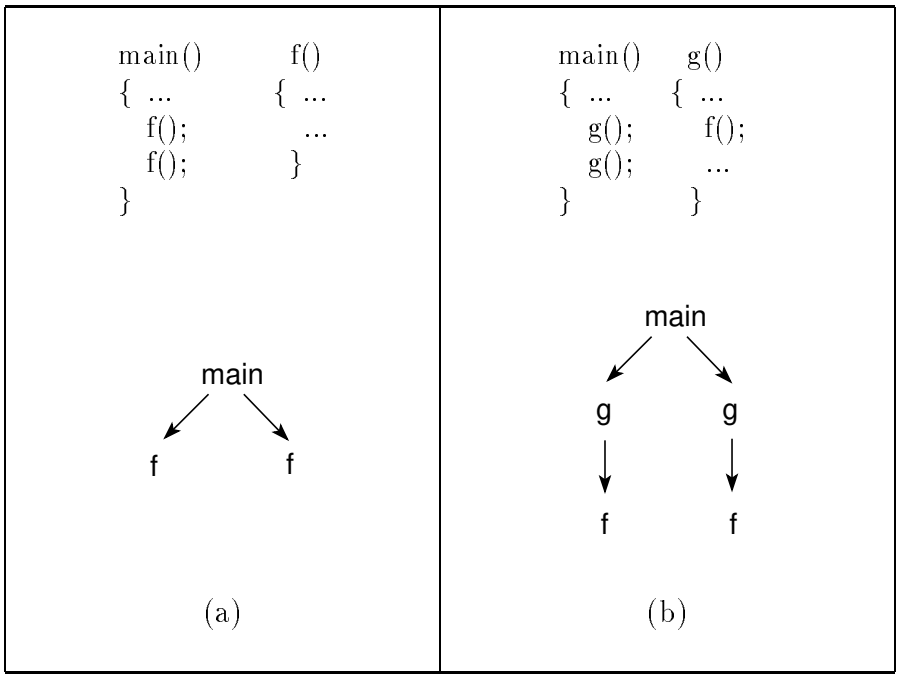


Figure 2.7: Invocation Contexts

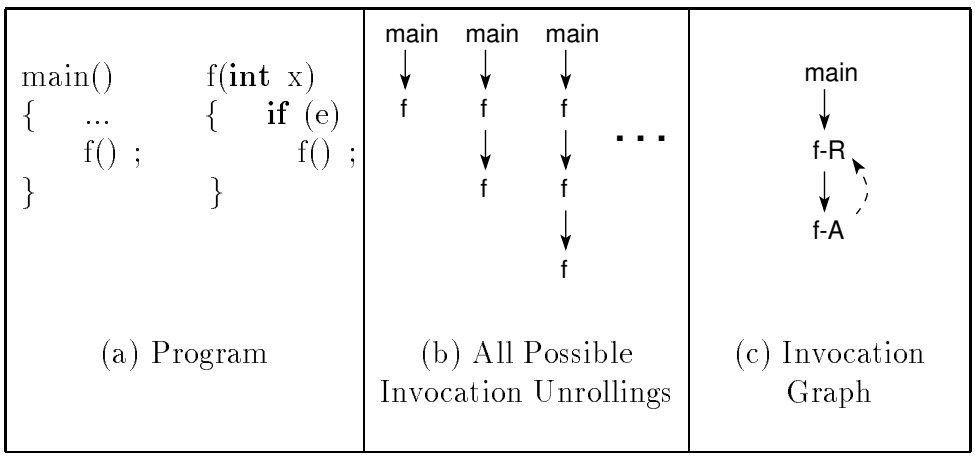


Figure 2.8: Invocation Graph for Recursion

The approach of explicitly building the invocation graph has the following advantages: (1) it cleanly separates the abstraction for any interprocedural analysis from the abstraction required to encode the calling context, (2) it allows one to deposit information computed from one analysis that can be useful for the next analysis, (3) it provides a place to store (memoize) IN/OUT pairs previously computed to summarize the effect of the function call (so that extra computation can be avoided at analysis time), and (4) it provides a simple framework for implementing simple compositional fixed-point computations for recursion.

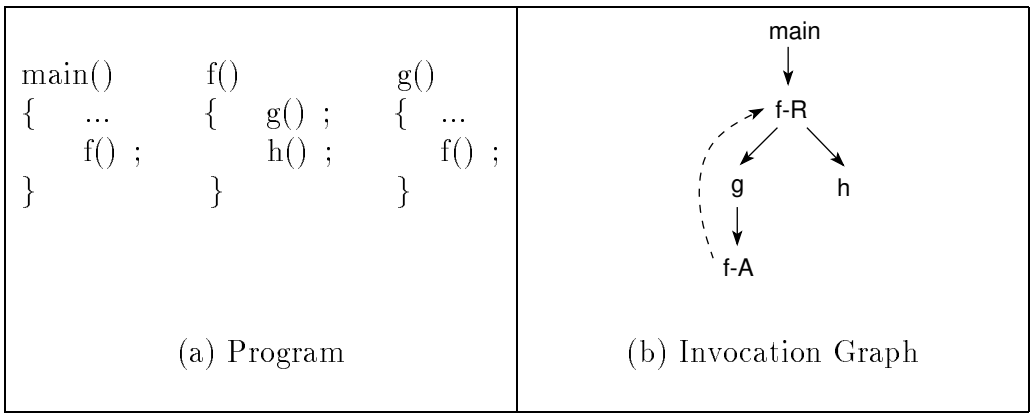


Figure 2.9: Invocation Graph for Mutual Recursion

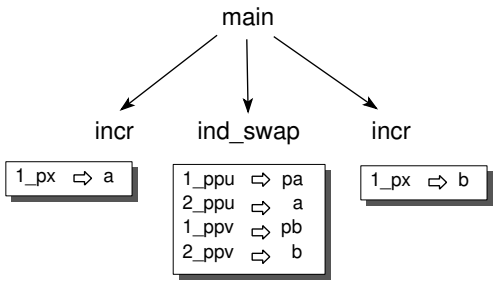
2.4.2 Map Information in the Invocation Graph

In the presence of procedure calls, an indirect reference in a procedure can refer to variables that are outside its scope (henceforth termed as *invisible* variables). This can happen, for example, when the address of a local variable is passed as a parameter, or when a global pointer points-to local variables of the caller. As each *accessible* real stack location needs to be represented by a *named* abstract stack location, points-to analysis generates special symbolic names to represent such invisible variables. A symbolic name is generated for each possible level of indirection of formal parameters and global pointers. Next each invisible variable is *mapped* to a unique symbolic name.

For example in Figure 2.10, for the formal parameter `ppu` with type `int**`, symbolic names `1_ppu` and `2_ppu` with types `int*` and `int` are generated. Now, since the

```
void ind_swap(int **ppu, int **ppv)
{ int  *ptemp;
  ptemp = *ppu;
  *ppu = *ppv;
  *ppv = ptemp;
}

void incr(int *px)
{ int temp_0;
  temp_0 = *px;
  *px = temp_0 + 1;
}

int main()
{ int a; int b; int *pa; int *pb;
  int **temp_2; int **temp_1;
  a = 1; b = 10;
  pa = &a; pb = &b;
  incr(pa);
  temp_1 = &pa; temp_2 = &pb;
  ind_swap(temp_1,temp_2);
  incr(pa);
}
```

(a) Example Program



(b) Invocation Graph and Map Information

Figure 2.10: Map Information

address of the local variable `pa` (invisible to `ind_swap`) is passed to `ppu`, the symbolic name `1_ppu` is used to represent `pa` in the procedure `ind_swap`. Similarly, as the invisible variable `a` is accessible through the indirect reference `**ppu`, it gets represented by the symbolic name `2_ppu`. This association of invisible variables with symbolic names is recorded in the invocation graph nodes as *map* information. Figure 2.10(b) shows the map information for various procedure calls. Complete details of the mapping process can be found in [Ema93].

The map information is context-sensitive as can be seen from the different mappings for the two calls to procedure `incr` in Figure 2.10(b). The symbolic names are independent of the context. Points-to analysis and other interprocedural analyses use the symbolic names in a context-free manner when analyzing a procedure. On returning from a procedure, they are *unmapped* to appropriate variables based on the map information recorded in the invocation graph, for the given calling context.

2.4.3 Resolving Function Pointers

In C, pointers may not only point to memory locations, but also to functions. This means that the complete invocation graph cannot be built by a simple textual pass over the program. Thus the points-to analysis must complete the invocation graph by resolving which functions are invokable from each indirect function pointer call. One might ask why the completion of the invocation graph must proceed at the same time as the points-to analysis. Consider that the complete invocation graph cannot be built before points-to analysis because the meaning of an indirect call `(*pf)()` is determined by examining the objects that `pf` may point-to. However, points-to analysis itself is a context-sensitive interprocedural analysis that needs the invocation graph. Thus the two approximations must be calculated at the same time.

To explain how points-to analysis completes the invocation graph, we give an example. Consider the program in Figure 2.11(a), with an indirect call `fp()` inside the while loop. The (incomplete) invocation graph of the program before points-to analysis is given in Figure 2.11(b). When points-to analysis encounters the indirect call `fp()`, during the first iteration of the while-loop fixed-point computation, it finds the current points-to set of `fp` to be `(f)`. The invocation graph is updated according to this information, as shown In Figure 2.11(c) and the function `f` is analyzed in the current calling context. During the second iteration of the fixed-point computation,

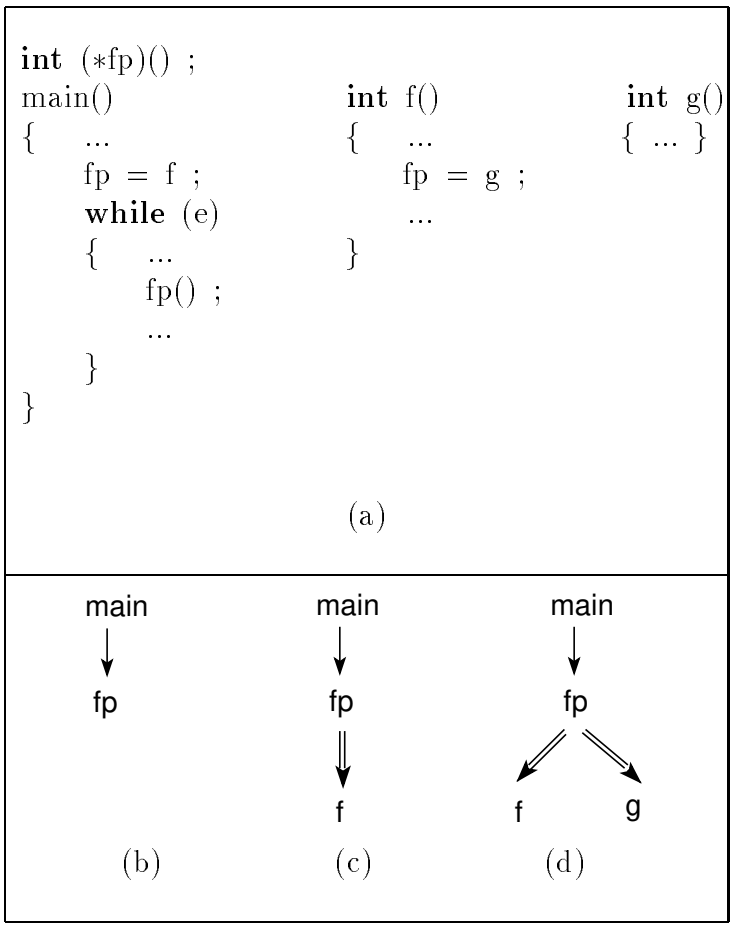


Figure 2.11: Invocation Graph for Function Pointers

the points-to set of `fp` becomes `(f,g)`, and the invocation graph again gets updated as shown in Figure 2.11(d). Future iterations do not modify the points-to set of `fp`, and thus the points-to analysis constructs the complete invocation graph for other interprocedural analyses. The complete algorithm for resolving function pointers can be found in [Ghi92, EGH94].

Other interprocedural analyses can measure the effect of a function pointer call, by merging the outputs obtained by individually analyzing in the current calling context, all the functions represented by its child nodes in the invocation graph.

2.5 Computing Stack Read/Write Sets

The goal here is to decorate each statement in the *simplified* program with the sets of abstract stack locations (ASLs) it reads and writes. A hierarchical approach is followed, where read/write sets are computed for not only basic SIMPLE statements but also for compositional control statements and functions. The sets for the latter two categories are obtained by unioning the sets of their constituent statements.

It is straightforward to compute read/write sets for basic SIMPLE statements that only involve scalars (types 1, 3, 5, 9 and 11 in Figure 2.2). For example, a statement of type `x = a binop b` reads the ASLs `a` and `b` and writes the ASL `x`. For statements involving indirection (types 2, 4, 6, 8, 10, 12, 14), one also needs to factor in points-to information. For example, consider the statement `S: *p = *q` with points-to information {`(p, x, D)`, `(q, b, P)`, `(q, heap, P)`}. As per its syntax, the statement reads the ASLs `q` and `p`. Factoring in the points-to info, it also reads the ASLs `b` and `heap`, and writes the ASL `x`, giving the following final information: Read(S) = {`p, q, b, heap`} and Write(S) = {`x`}. In the McCAT implementation, read/write sets are further subdivided into may and must sets indicating which locations are may/must read/written.

For statements that involve function calls (types 7, 8 and 15 in Figure 2.2), an interprocedural computation is required. The main idea is that read/write sets for each function are computed in terms of its parameters, symbolic invisible variables representing local variables of the callers, and global variables. The sets for a function call are computed from the sets of the callee function by appropriately *unmapping* the symbolic variables of the callee to local variables of the caller.

Consider for example the program shown in Figure 2.10. Inside function `incr`, the pointer `px` definitely points-to the symbolic variable `1_px`. Based on this information, the first statement of `incr` reads the ASL `1_px` and its second statement writes the ASL `1_px`, giving Read(incr) = {`px, 1_px`}, and Write(incr) = {`1_px`} (the local variable `temp_0` is not included in function level read/write sets). For the first call to `incr` inside `main`, `1_px` represents the local variable `a` (Figure 2.10(b)). So the read/write sets for this call are respectively computed as {`pa, a`} and {`a`}. Similarly, the read/write sets for the second call are respectively computed as {`pa, b`} and {`b`}, as `1_px` represents the local variable `b` in this context.

It is possible that a given call-site appears on more than one call-chain, i.e., is represented by more than one invocation graph node. Thus it can have more than one set of map information associated with it. To accurately use map information, the interprocedural computation of read/write sets is also done in a context-sensitive manner, by following the invocation graph built by points-to analysis. On encountering a function call, first the read/write sets for the function under the current invocation context are computed. Next, these sets are unmapped according to the map information deposited in the current invocation graph node corresponding to the procedure call. This scheme ensures that read/write sets for a function call do not contain locations accessed by the callee function in invocation contexts not relevant for the given call.

2.6 Heap Analyses

The abstractions for heap analysis, implemented in the McCAT compiler, are variations on the path matrix model of Hendren and Nicolau [HN90]. The path matrix approach captures the structure of the heap using an abstraction, orthogonal to the k-limited graphs. It essentially exploits the fact, that though there are potentially infinite number of objects in the heap, they are always accessed using access-paths which *originate* from *stack-resident heap-directed* pointers. Figure 2.12 provides an illustration of this observation. It can be easily noted, that access to any node of the data structure built in the heap, has to originate from one of the pointer variables on the stack (`p`, `q`, `r`, `s`, `t`). They term such *heap-directed* pointers as *handles*, as these are the pointers the programmer has handle on. Since *handles* are themselves resident on the stack, they are finite in number, and are relatively inexpensive to reason about.

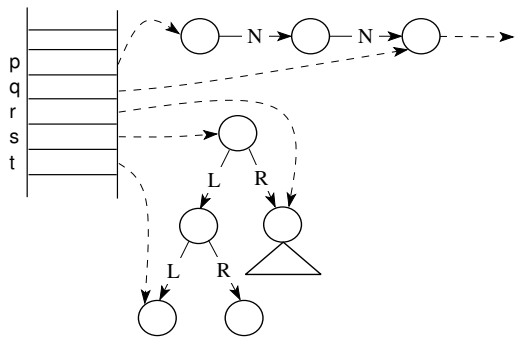
Based on the above observations, the abstraction developed by Hendren and Nicolau, is a matrix of *handles* `P`, where an entry `P[p,q]`, contains the access path inside the heap, from handle `p` to handle `q`. The access paths are expressed in the form of restricted regular expressions. Figure 2.12(b) presents an example of how the heap structure is captured using the path matrix. An empty entry, say `P[r,q]` indicates that the heap object pointed to by `r` cannot be reached by using an access path originating from pointer `q` and vice versa. The symbol `S` in an entry denotes that the two pointer variables point to the same heap object. The more complicated expressions, represent the access paths using the link fields.

However, the path matrix analysis assumes and verifies that the underlying data structure being created and manipulated by the program is a tree. The path matrix information is used to distinguish between pointers accessing disjoint subpieces of the tree, and to detect the creation and elimination of temporary DAG nodes during tree updates. It cannot handle cyclic structures. Further, the encoding of precise path relationships in the form of path expressions, can prove to be potentially expensive information to store.

The McCAT approach provides a more practical and general purpose heap analysis (without any assumptions about the underlying data structures). It also follows the paradigm of abstracting the heap structure by capturing the relationships only between the *handles*. However, McCAT heap analyses capture coarser path relationships, which can be stored as boolean matrices. This enables faster data flow merge operation and substantially reduces the storage requirements for the analyses, while the analyses still gather *useful* information. Thus the first heap analysis, *connection analysis* [GH96a] only abstracts the information, if two handles point to nodes in the same data structure, using a connection matrix.

The second heap analysis, *shape analysis* [GH96b], abstracts the boolean information if a path *exists* from the node pointed by one handle to the node pointed by another handle (as opposed to the actual path expression). Additionally it keeps track if two given handles can access a common node using an interference matrix, and also abstracts the shape of the data structure *accessible* from a given handle as a "Tree", "DAG" or "Cyclic Graph".

Figure 2.12 also illustrates the above matrices for the heap structure shown in part (a) of the figure. Part (c) shows the connection matrix, parts (d) and (e) show the direction and interference matrices used for shape analysis, while part (f) shows



(a) Heap Structure

	p	q	r	s	t
p	S	N^2			
q		S			
r			S		
s			R	S	L^2
t					S

(b) Path Matrix

	p	q	r	s	t
p	1	1	0	0	0
q	1	1	0	0	0
r	0	0	1	1	1
s	0	0	1	1	1
t	0	0	1	1	1

(c) Connection Matrix

	p	q	r	s	t
p	1	1	0	0	0
q	0	1	0	0	0
r	0	0	1	0	0
s	0	0	1	1	1
t	0	0	0	0	1

(d) Direction Matrix

	p	q	r	s	t
p	1	1	0	0	0
q	0	1	0	0	0
r	0	0	1	0	0
s	0	0	1	1	1
t	0	0	0	0	1

(e) Interference Matrix

{p,q,r,s,t}.shape = Tree

(f) Shape Attribute

Figure 2.12: An Example Showing Various Heap Abstractions

the shape attribute for all the handles which is "Tree" in this case. One can note that the connection matrix shows pointers `p` and `q` as connected, but not pointers `p` and `s`. Further the direction matrix indicates that there is a path from `s` to `t` but not vice-versa.

The heap analyses are implemented on the structured tree-based SIMPLE intermediate representation. Points-to analysis builds a complete framework to perform them efficiently and accurately in an interprocedural fashion. The overall implementation setting for the analyses is shown in Figure 2.1, which can be briefly described as follows:

First, all the `.c` files for the given program are given as input to the source level linker, which generates the FIRST AST for the entire program. Subsequently this AST is *simplified* and structured (i.e. `goto` statements are eliminated). The invocation graph is then constructed by identifying the functions called by each function in the program. Next, points-to analysis is performed, which calculates possible pointer targets at each program point for stack-directed pointers, and also resolves indirect calls through function pointers. After points-to analysis, hierarchical stack read/write sets are computed. Using the read/write set information, SSA numbering is performed [LH96]. Subsequently various interprocedural heap analyses are conducted.

At this stage, several new components are added to the compilation framework to *put pointer analysis to work*. These components are shown in bold face in Figure 2.1, and are based on the work reported in this thesis. Using these additions, first heap read/write sets are computed. Both stack and heap read/write sets are then used to implement more effective scalar optimizations, and enable more accurate array dependence testing. The read/write set information is also used as a program understanding aid, displayed via a Web browser-based tool. Further, for programs using recursive data structures, *parallelization* opportunities are identified using shape analysis information. In the following chapters, we discuss these new components of the McCAT compilation framework in detail.

Chapter 3

Connection Analysis and Heap Read/Write Sets

In the last chapter, we described the computation of stack read/write sets based on points-to analysis. Stack read/write sets provide a very conservative approximation of the heap. In this chapter we develop techniques for computing sharper heap read/write set information with respect to a heap analysis. We have chosen connection analysis [GH96a], the first analysis in the McCAT hierarchy of *storeless* heap analyses, for this purpose. Connection analysis was chosen, because it is a relatively simple heap analysis, clearly illustrates the problems in computing read/write sets from a storeless analysis, and yields interesting results when "put to work" properly.

Connection analysis was previously designed and implemented in the McCAT C compiler [Ghi95]. Thus, the focus of this chapter is not on connection analysis itself. Rather, its emphasis is on new techniques that augment the existing connection analysis, and enable the accurate computation of heap read/write sets. The rest of this chapter is organized as follows. In section 3.1, we briefly review the connection analysis abstraction and the associated analysis rules. We identify the problems in computing connection-based heap read/write sets in section 3.2. In section 3.3, we introduce the concept of anchor handles to compute the heap read/write sets. In section 3.4, we provide implementation details for augmenting connection analysis with anchor handles. We evaluate the efficiency of our analyses with respect to a set of 16 C benchmark programs in section 3.5. Finally, in section 3.6, we prove the correctness of the anchor-based read/write sets, and summarize the chapter in section 3.7.

3.1 Connection Analysis

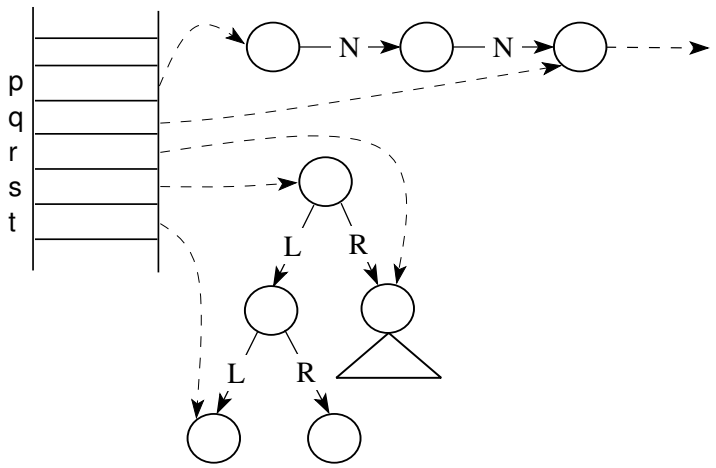
Connection analysis uses a *storeless* abstraction based on connection matrices. A *connection matrix* C is a boolean matrix of relationships between heap-directed pointers which captures simple connectivity of heap objects. A *heap object* is defined as a memory object allocated in the heap memory, representing an instance of a valid type definition (basic or user-defined) in the program. The connection matrix abstraction is designed to disambiguate heap accesses at the data structure level. The term *data structure* in this context represents a connected region in the heap, i.e. if the heap is viewed as an undirected graph with heap objects as nodes and links between them as edges, each connected component forms a separate data structure. Given any two data structures, they would not have a common heap object belonging to them. For example in Figure 3.1, the heap consists of two data structures: one pointed to by pointers `p` and `q` and the other pointed to by pointers `r`, `s`, and `t`.

With the above definitions, given any two heap-directed pointers say `p` and `q`, connection matrix abstracts the following *program-point-specific* relationships for a program point *P*:

- C[p,q] = 1 : Pointers `p` and `q` *possibly* point to heap objects belonging to the same data structure. In our terminology, pointers `p` and `q` are considered to be *connected*, or to have a *connection* relationship. This information is also denoted as *(p,q)* $\in$ Connect(P), where *(p,q)* is refered to as a connection pair.
- C[p,q] = 0 : The heap objects pointed to by pointers `p` and `q` *definitely* belong to different data structures. In other words, pointers `p` and `q` are *not* connected. This information is also alternatively presented as *(p,q)* $\notin$ Connect(P).

The useful information is the *negative* information. If pointers `p` and `q` are not connected, then heap accesses originating from them will always lead to disjoint heap locations, and thus not interfere. It is *safe* to report two heap-directed pointers to be connected, when they are not. However, if they can point to the same data structure, they should always be reported to be connected.

We illustrate the abstraction in Figure 3.1. Part (a) shows the structure of heap at a program point, while part (b) shows its abstraction as a connection matrix. We now briefly explain the entries in the connection matrix. The zero in the entry C[p,r] indicates that pointers `p` and `r` point to disjoint data structures in the heap. The



(a) Heap Structure

	p	q	r	s	t
p	1	1	0	0	0
q	1	1	0	0	0
r	0	0	1	1	1
s	0	0	1	1	1
t	0	0	1	1	1

(b) Connection Matrix Abstraction

Figure 3.1: An Example Connection Matrix

one in the entry C[s,r] indicates that `s` and `r` point to objects belonging to the same data structure. Note that the entry C[r,t] is set to one, despite the fact that pointers `r` and `t` point to disjoint subpieces of the same data structure. This is because connection analysis is designed to disambiguate heap accesses at the data structure level (for efficiency reasons). To do more fine-grain disambiguation, one can use more sophisticated analyses like shape analysis [GH96b], which can distinguish between subpieces of a data structure itself. This is the topic of chapter 5.

The following are some other important characteristics of the connection matrix abstraction:

- It abstracts relationships only between *stack-resident* heap-directed pointers. As all heap accesses originate from these pointers, their relationships effectively capture the structure of the heap. For example in Figure 3.1(b), the information that pointers `p` and `s` point to disjoint data structures also implies that pointers `p->N` and `s->L` point to disjoint structures.

- For each function in the program, the connection matrix abstracts relationships between all stack-resident pointers which can be heap-directed at some point in the program and are directly or indirectly (through an indirect reference) accessible from the function. Names are naturally available from the program, for directly accessible pointers. For indirectly accessible pointers, special symbolic names are generated by points-to analysis as explained in section 2.4.2. These names are reused by connection analysis. To know which pointers ever point to heap, the existing points-to information is used.

- If a pointer, say `p`, does not point to a heap location at a given program point, the connection matrix entry C[p,p] is set to zero at that program point. In this case the pointer points to `NULL` or to a stack location.

- The connection matrix relationship is symmetric, i.e. for any two heap-directed pointers say `p` and `q`, we always have C[p,q] = C[q,p]. The connection relationships shown in Figure 3.1(b) illustrate this property. It is used in the actual implementation to reduce the storage requirement by half.

3.1.1 Illustrative Examples

In this section, we briefly illustrate the rules to compute connection relationships. The complete details can be found in [Ghi95, GH96a].

Let us follow the computation of connection relationships for the code segment shown in Figure 3.2(a). At program point `S` (just before the statement `S`), pointers `p`, `q`, and `r` point to disjoint (single node) data structures, so we have the set Connect(S) = {} (Figure 3.2(b)). The statement `S: p->next = q`, *connects* the data structures pointed to by `p` and `q`. Thus after statement `S`, `p` and `q` point to the same data structure, giving the following connection set at program point `T`: Connect(T) = {*(p,q)*} (Figure 3.2(c)).

The statement `T: p->next = r` in Figure 3.2(a), involves both a *connection* and a *disconnection*. First, the destructive update to the field `next` disconnects the data structures pointed to by `p` and `q`. Next, the assignment of `r` connects the data structures pointed to by `p` and `r`. The disconnection should kill the connection pair *(p,q)*. However, connection analysis does not have sufficient information to *safely* kill this connection pair. It is possible that the two pointers are also connected via another link, say `forward`. In this case, the pointers will remain connected even after updating the `next` link.

Connection analysis only abstracts the boolean relationship whether two heap pointers are connected. It does not collect the additional information about the particular link fields through which the pointers are connected, for efficiency reasons. Thus it cannot infer the above kill information, and errs conservatively, not killing any connection pairs.

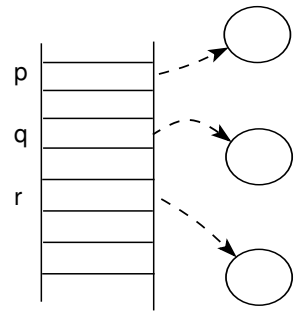
The statement `T` generates new connection relationships, due to the interlinking of data structures pointed to by `p` and `r`. All pointers connected with `p` (which include `q`), now get connected with all pointers connected with `r`. Thus we get the following connection set at program point `U`: Connect(U) = {*(p,q), (p,r), (q,r)*}. Note that the analysis generates the spurious connection pairs *(p,q)* and *(q,r)*, due to conservative kill information.

Finally, the statement `U: q = NULL`, kills all the connection relationships with respect to `q`, giving the following connection set at program point `V`: Connect(V) = {*(p,r)*} (Figure 3.2(d)). Note that the analysis is able to obtain precise kill information, when the statement updates a stack-based heap pointer (`q`), as opposed to when a heap-resident heap pointer (`p->next`) is updated.

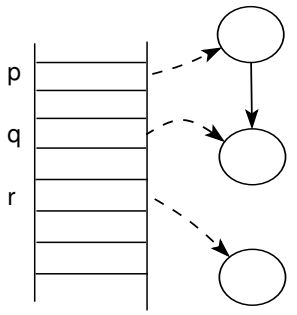
Above, we have illustrated the basic principles of connection analysis. The analysis has been implemented as a context-sensitive interprocedural analysis in the McCAT C compiler. It is performed on the SIMPLE intermediate representation. The implementation of connection analysis is structured as a simple analysis for each basic statement, a compositional rule for each control construct, and an interprocedural strategy that uses the invocation graph to capture all calling contexts.

```
    p = malloc();
    q = malloc();
    r = malloc();
S:  p->next = q;
T:  p->next = r;
U:  q = NULL;
V:
```

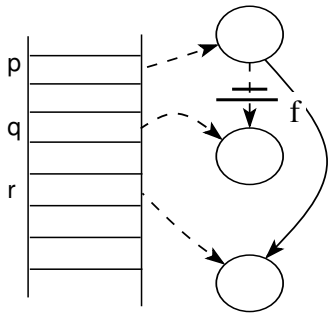
(a) Code Segment being Analyzed



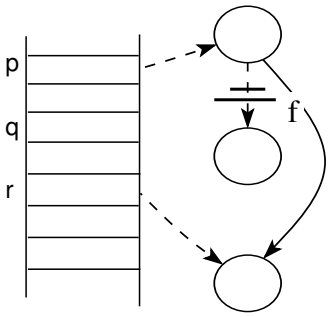
(a) program point S
Connect(S) = {}



(b) program point T
Connect(T) = {*(p,q)*}



(c) program point U
Connect(U) =
{*(p,q), (p,r), (q,r)*}



(d) program point V
Connect(V) = {*(p,r)*}

Figure 3.2: Demonstrating Connection Analysis

The overall structure of connection analysis for basic statements is shown in Figure 3.3. Part(a) gives the complete list of the basic statements that can affect connection relationships. Here, variables `p` and `q` and the field `f` are of pointer type, variable `k` is of integer type, and `op` denotes the `+` and `-` operations. Part(c) shows how the new connection matrix C_n at program point `y`, is obtained from the kill and gen sets for the statement at program point `x`, with the input matrix as C. Here, H denotes the set of heap-directed pointers. The kill and gen sets for each basic statement are summarized in Table 3.1. Connection information for all basic SIMPLE statements (Figure 2.2 on page 28) can be computed via a composition of the rules shown in Figure 3.1. Detailed descriptions can be found in [Ghi95, GH96a].

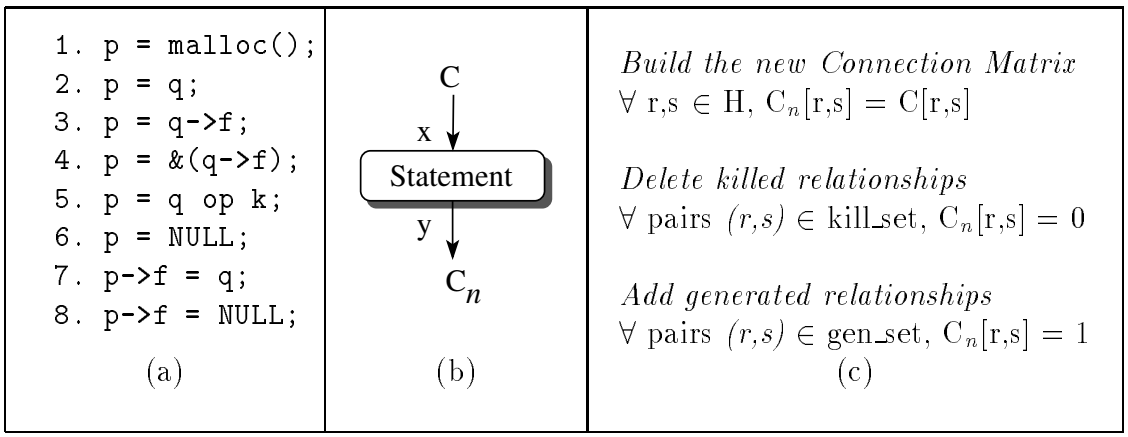


Figure 3.3: Computing Connection Matrix C_n from C for Basic Statements

For compositional control statements in SIMPLE, the analysis computes a fixed-point by *merging* connection matrices at control flow merge points. Two matrices are merged by applying the logical OR operator on each individual entry. Given two matrices C_a and C_b arriving at a control flow merge point, the new matrix C_n is computed using the following rule:

$$\forall \text{ r,s} \in \text{H, } C_n[\text{r,s}] = C_a[\text{r,s}] \wedge C_b[\text{r,s}]$$

The overall strategy for interprocedural analysis is depicted in Figure 3.4 The general idea is that, first, the connection matrix C_m at the call-site is *mapped* to prepare the input connection matrix C_e for the called procedure. Next, the body of the procedure is analyzed with this input matrix and the output matrix obtained (C_x) is *unmapped* and the resulting matrix C_n is returned to the call-site. The *map*

Statement	Kill Set	Gen Set
`p = malloc()`	{ *(p,s)* | C[p,s] }	{ *(p,p)* }
`p = q`	{ *(p,s)* | C[p,s] }	{ *(p,s)* | C[q,s] } ∪ { *(p,p)* | C[q,q] }
`p = q->f`	{ *(p,s)* | C[p,s] }	{ *(p,s)* | C[q,s] } ∪ { *(p,p)* | C[q,q] }
`p = &(q->f)`	{ *(p,s)* | C[p,s] }	{ *(p,s)* | C[q,s] } ∪ { *(p,p)* | C[q,q] }
`p = q op k`	{ *(p,s)* | C[p,s] }	{ *(p,s)* | C[q,s] } ∪ { *(p,p)* | C[q,q] }
`p = NULL`	{ *(p,s)* | C[p,s] }	{}
`p->f = NULL`	{}	{}
`p->f = q`	{}	{ *(r,s)* | C[p,r] ∧ C[q,s] }

Table 3.1: Basic Connection Analysis Rules

process involves generating special symbolic names to represent out-of-scope pointers in the context of the callee function. The *unmap* process computes the connection relationships of variables after the function call, from the connection relationships of the variables they are *mapped* to in the callee. The overall interprocedural framework for connection analysis is similar to that described for points-to analysis in section 2.4. The complete details of the implementation can be found in [Ghi95, GH96a].

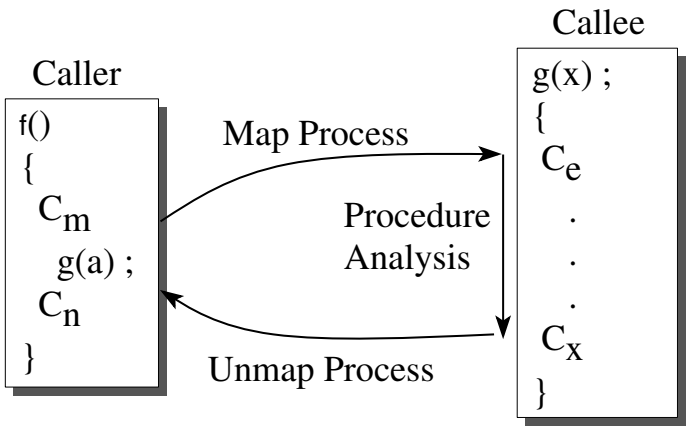


Figure 3.4: Interprocedural Startegy

3.2 Computing Read/Write Sets Based on Connection Analysis

Our main goal is to identify the set of locations read/written by a given statement or program region. We have already shown how to compute accurate stack read/write sets even in the presence of indirect pointer references, using points-to information (section 2.5). We noticed that points-to based stack read/write sets give very conservative approximation for statements involving indirect references to the heap. We now focus on obtaining sharper read/write set information for heap references, based on connection analysis results.

Difficulties in Computing Connection-based Read/Write Sets

Since connection analysis is storeless, it is difficult to relate connection information at different program points. We illustrate the problem, using the example program shown in Figure 3.5. Consider the function `flip` in the figure. At call-site `D` for function `flip`, pointers `s` and `t` point to disjoint heap objects and are not connected. Consequently, at the entry to `flip` parameters `a` and `b` are not connected, and they remain disconnected over the entire function body (no statement connects them). Suppose we want to summarize the locations read and written relative to the parameters `a` and `b`. Based on the connection information we may deduce that the sets Write(S) = {`a->x`}, and Read(T) = {`b->x`} do not conflict, as `a` and `b` point to disjoint objects at both `S` and `T`.

However, such simplistic solutions do not suffice. Consider the function, `tricky_flip`, given in Figure 3.6(a). As illustrated in Figures 3.6(b) and 3.6(c), the pointer values stored in `a` and `b` change. Thus, even though `a` and `b` are not connected at point `S`, and also not connected at point `T`, there is a flow dependence since `a->x` at point `S` is the same location as `b->x` at point `T`.

3.3 Dropping Anchors

The fundamental problem in the `tricky_flip` example is that we don't have enough names for heap objects. The same programmer-defined name may refer to different objects at different program points. The solution is to invent enough new names, *anchor handles*, so that further analyses can be done simply, based on information relative to the anchor handles. Consider a new version of `tricky_flip` as given in Figure 3.7. In this example, we have introduced the names `a@tricky_flip` and

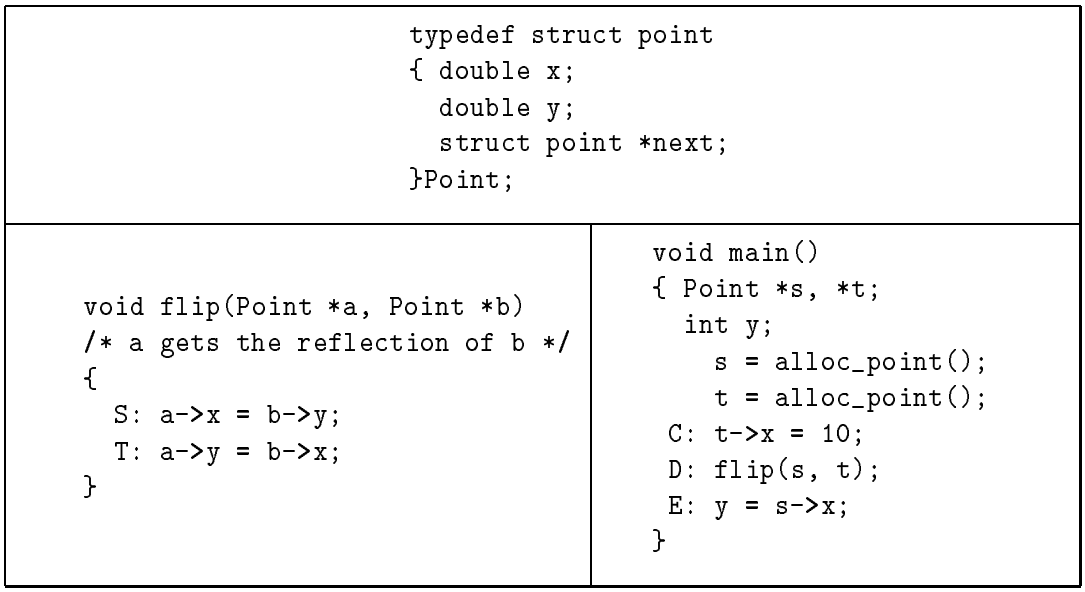


Figure 3.5: Example for Connection-based Read-Write Sets

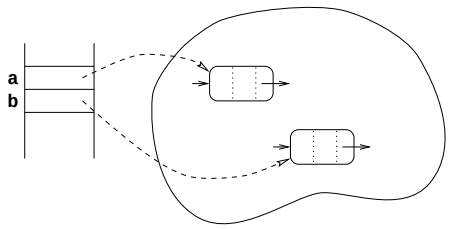
`b@tricky_flip` which anchor the parameters, and we have introduced `a@S`, which is an anchor for variable `a` at program point `S`. Anchor handles are given their initial values by introducing *ghost copy statements*. All anchors for parameters are copied at the beginning of a function body, while program-point-specific anchors of the form `a@S` are copied at the given program point `S`.[1] Although we have shown these as part of the program, in fact, the anchors and ghosts are only implemented in the connection analysis, and the actual program is not modified.

Anchor handles serve as "anchor points" for analysis within the body of a function. Informally, we can now compute the read/write sets with respect to heap-related indirect references by noting that an anchor handle `x@p` is read/written each time any pointer connected to `x@p` is read/written. For example, as illustrated in Figure 3.7, at statement `S`, `a` is connected with anchors `a@tricky_flip` and `a@S`, and so we would indicate that writing `a->x` actually writes to the anchored locations `a@tricky_flip->x` and `a@S->x` giving HeapWrite(S) = {`a@tricky_flip->x, a@S->x`}. Similarly, at statement `T`, `b` is also connected with anchors `a@tricky_flip` and `a@S`, so the read `b->x` actually reads the anchored locations `a@tricky_flip->x` and `a@S->x` giving HeapRead(T) = {`a@tricky_flip->x, a@S->x`}. Comparing the sets HeapWrite(S) with HeapRead(T), one can detect the flow dependence from `S` to `T`.

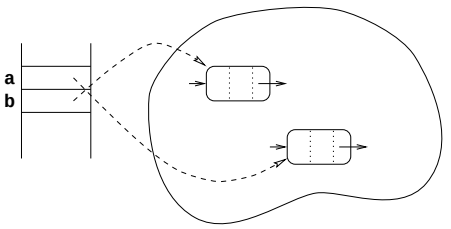
[1]For the sake of clarity we do not include other anchors like `b@S` and `a@T` in this example.

```
void tricky_flip(Point *a, Point *b)
{ Point *tmp;

S: a->x = b->y;
   /* swap a and b */
R: tmp = a; a = b; b = tmp;
T: a->y = b->x;
}
```

(a) `tricky_flip`



(b) program point `S`, connect(S) = {}

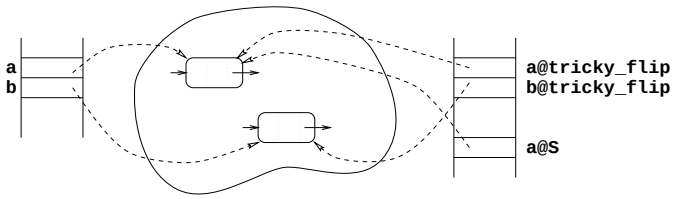


(c) program point `T`, connect(T) = {}

Figure 3.6: Tricky Example

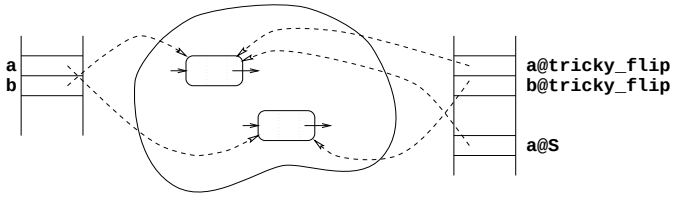
```
void tricky_flip(Point *a, Point *b)
{ Point *tmp;
  Point *a@tricky_flip, *b@tricky_flip, *a@S; /* anchors */

   a@tricky_flip = a; b@tricky_flip = b; a@S = a; /* ghosts */
S: a->x = b->y;
   /* swap a and b */
   tmp = a; a = b; b = tmp;
T: a->y = b->x;
}
```

(a) `tricky_flip`



$$\text{connect(S)} = \{(a, a@tricky_flip), (b, b@tricky_flip), (a, a@S), (a@S, a@tricky_flip)\}$$

(b) program point `S`



$$\text{connect(T)} = \{(b, a@tricky_flip), (a, b@tricky_flip), (b, a@S), (a@S, a@tricky_flip)\}$$

(c) program point `T`

Figure 3.7: Dropping Anchors and Inserting Ghost Copy Statements

We can also collect heap read/write sets with respect to function parameters in terms of their anchors. In this example, we have HeapWrite(tricky_flip) = HeapRead(tricky_flip) = {`a@tricky_flip->x, b@tricky_flip->y`}. This function level information could be used to determine that at the entry of function `tricky_flip`: (1) it might be useful to prefetch `a->x` and `b->y`, but one should not prefetch `b->x` and `a->y`; and (2) there are no updates to the `next` field of either `a` or `b`, and thus `tricky_flip` does not change the "listness" of any data structure.

3.4 Introducing Anchors in Connection Analysis

We have augmented our implementation of connection analysis to include anchor handles and appropriate ghost copy assignments, and to compute heap read/write sets with respect to the anchors. In this section, we first describe our strategy for generating anchor handles (subsection 3.4.1). We then discuss how these handles are introduced into the analysis via ghost copy assignments, and how they are used to compute read/write sets for basic statements (subsection 3.4.2), control statements (subsection 3.4.3), and function calls (subsection 3.4.4).

3.4.1 Generating Anchor Handles

Our implementation includes a preprocessing pass through the program, which generates the required anchor handles and associates them with appropriate statements. For a given function in the program, anchor handles are generated for the following:

1. Heap-directed formal parameters.
2. Heap-directed global pointers accessed in the function,
3. Call-sites that read/write heap locations.
4. Heap-related indirect references in the function body.

A pointer `p` is considered heap-directed for a function, if the entry `(p, heap, P)` exists in the points-to set at *some program point* in the function. Similarly, an indirect reference `*p` is considered heap-related if the entry `(p, heap, P)` is in the points-to set at the *given program point*.

The first two types of anchor handles are introduced to compute heap read/write sets for the entire function. The anchor handles `a@tricky_flip` and `b@tricky_flip` in Figure 3.7, fall into this category. These handles are associated with the function header itself. Call-site anchor handles are used to compare heap read/write sets of a function call with those of other statements. For a given call-site `C`, that reads/writes heap locations, two anchor handles, namely `readAnchor@C` and `writeAnchor@C`, are generated. The need for these handles is discussed in section 3.4.4. For basic statements with indirect references to heap, anchor handles are generated with respect to the particular heap-directed pointers dereferenced. For example, for the statement `S: q->x = p->y`, two anchor handles, namely `p@S` and `q@S`, are generated. Further, the handles `p@S` and `q@S` are respectively associated with the statement as its read and write anchor handles.

Note that the points-to information about which indirect references/function calls can access the heap, enables us to reduce the number of anchor handles generated. It enables us to determine which pointers can point to the heap, and ignore the pointers that can only point to stack locations. We use extended SSA numbers [LH96] to further reduce the number of anchor handles required. Although conceptually one requires a new anchor handle for each indirect reference to the heap, in fact, anchor handles can often be reused. For example, in the program in Figure 3.7 the anchor `a@tricky_flip` can also be used as the anchor `a@S`, because they anchor the same location. In general, the same handle can be used to anchor all indirect references involving a given definition of a pointer. Extended SSA numbering associates a new primary SSA number with a variable (including pointers), whenever it is potentially updated (including indirect updates). Thus, when we generate an anchor handle for a pointer `ptr`, we also associate its current SSA number `k` with the handle. If an anchor handle for `ptr` with the SSA number `k` already exists we do not generate a new handle, and reuse the existing one for the given indirect reference.

Field Anchors: For introducing anchors, the case like (`S: r = &(q->item); T: *r = x;`) is specially handled. Here statement `T` actually writes the field `item` with respect to the anchor `q@S` i.e. it writes the anchored location `q@S->item`. This is not evident from the program syntax. To handle this case, we connect `r` with a special anchor handle `q@S_item` at `S`. And from the presence of the pair *(r,q@S_item)* in the set Connect(T), we can correctly compute the set HeapWrite(T). These anchors are considered global in scope, as they contain additional field information, which would otherwise be lost during mapping/unmapping information at procedure calls. As they anchor specific fields inside objects, we term these anchors as *field anchors*.

3.4.2 Augmented Connection Analysis for Basic Statements

Once anchor handles are generated and associated with appropriate statements, the augmented connection analysis simultaneously computes connection information and connection-based heap read/write sets. The overall algorithm for the augmented connection analysis of a basic statement is outlined in the function *augmentedConnectionBasic* in Figure 3.8.

The augmented connection analysis of a basic statement proceeds in the following three steps:

1. **Ghost Copy Assignments:**

   First, ghost copy assignments are performed to include the statement anchors in the connection set, via the function *ghostCopyAssign*. This function checks if a read/write anchor is associated with the statement, and then performs connection analysis of the ghost copy assignment involving the anchor and the corresponding anchored pointer, as shown in Figure 3.8. As per the connection analysis rule for a simple assignment statement (statement `p = q` in Table 3.1), first all the connection relationships of the anchor are killed, and next it is connected with all the pointers currently connected with the anchored pointer.

2. **Heap Read/Write Set Computation:**

   After performing the ghost copy assignments, heap read/write sets are computed with respect to the statement anchors. The algorithm is outlined in Figure 3.9. For each field, say `field`, read/written by the statement, the corresponding anchored location `stmtReadAnchor->field/stmtWriteAnchor->field` is included in the heap read/write set of the statement. Additionally, for each anchor, say `otherAnchor`, connected with `stmtReadAnchor/stmtWriteAnchor`, the anchored location `otherAnchor->field` is also included in the heap read/write set. Finally, to accurately identify the fields accessed, the function *getFieldSet* also checks for connections with *field anchors*, and includes the fields represented by them.

3. **Connection Analysis:** The last step of augmented connection analysis, performs connection analysis of the statement itself, to obtain the new connection set valid after the statement. The statement is analyzed as per the analysis rules shown in Table 3.1.

```
/* analyze the stmt with current connectionSet to get the
 * new connection set and the heap read/write sets */
fun augmentedConnectionBasic(stmt, connectionSet) =
  /* perform ghost copy assignments to get a new connection set */
  postCopyConnectionSet = ghostCopyAssigns(stmt, connectionSet);
  /* compute heap read/write sets using the post copy connection set */
  [readSet,writeSet] = basicHeapRWSets(stmt, postCopyConnectionSet);
  /* compute the new connection set valid after the stmt */
  newConnectionSet = connectionAnalyze(stmt, postCopyConnectionSet);
  return [newConnectionSet,readSet,writeSet];

/* perform ghost copy assignments for the read and write anchors */
fun ghostCopyAssigns(stmt, connectionSet) =
  newConnectionSet = ghostCopyAssign(stmt, connectionSet, READ);
  newConnectionSet = ghostCopyAssign(stmt, newConnectionSet, WRITE);
  return newConnectionSet;

/* perform ghost copy assignment for a given anchor */
fun ghostCopyAssign(stmt, connectionSet, accessType) =
  if (accessType == READ)
    stmtAnchor = getAnchor(stmt, READ); /* get read anchor */
    Ref = getRef(stmt, rhs); /* ref on rhs of the stmt */
  else /* if (accessType == WRITE) */
    stmtAnchor = getAnchor(stmt, WRITE); /* get write anchor */
    Ref = getRef(stmt, lhs); /* ref on the lhs of stmt */
  if (stmtAnchor != NULL)
    Var = getBaseVar(Ref); /* base var for the ref: for p->next it is p */
    ghostCopyStmt = [stmtAnchor = Var];
    newConnectionSet = connectionAnalyze(ghostCopyStmt, connectionSet);
  return newConnectionSet;
```

Figure 3.8: Augmented Connection Analysis For Basic Statements

```
/* compute heap read/write sets for basic stmts */
fun basicHeapRWSets(stmt, connectionSet) =
  readSet = basicHeapRWSet(stmt, connectionSet, READ);
  writeSet = basicHeapRWSet(stmt, connectionSet, WRITE);
  return [readSet,writeSet];

/* get the heap read or the heap write set for stmt depending on accessType */
fun basicHeapRWSet(stmt, connectionSet, accessType) =
  set = {}; /* initialize the read or write set to empty */
  if (accessType == READ)
    stmtAnchor = getAnchor(stmt, READ); /* get read anchor for read sets */
  else /* if (accessType == WRITE) */
    stmtAnchor = getAnchor(stmt, WRITE); /* get write anchor */
  if (stmtAnchor == NULL) /* does not read or write to heap */
    return set;
  /* set of fields read/written at stmt */
  fieldSet = getFieldSet(stmt, stmtAnchor, connectionSet, accessType);
  foreach anchor s.t. (anchor,stmtAnchor) ∈ connectionSet
    foreach field ∈ fieldSet /* fields read or write accessed at the stmt */
      set = set ∪ {anchor->field}; /* include anchored location in set */
  return set;

/* get the set of fields read or write accessed at the stmt */
fun getFieldSet(stmt, stmtAnchor, connectionSet, accessType) =
  fieldSet = {}; /* initialize the set of fields accessed at the stmt */
  if (accessType == READ)
    ref = getRef(stmt, rhs);
  else /* if accessType == WRITE) */
    ref = getRef(stmt, lhs);
  /* for ref p->next, next is included in fieldSet */
  fieldSet = fieldsAccessed(ref);
  foreach anchor s.t. (anchor,stmtAnchor) ∈ connectionSet
    if (isFieldAnchor(anchor)) /* anchors a field access: of type q@S_next */
      fieldSet = fieldSet ∪ fieldsAccessed(anchor); /* include next field */
  return fieldSet;
```

Figure 3.9: Computing Heap Read/Write Sets for Basic Statements

3.4.3 Augmented Connection Analysis for Control Statements

Compositional control statements in SIMPLE(`for`, `if`, `while` etc.), are handled by augmented connection analysis, in a fashion similar to the original connection analysis [Ghi95, GH96a] (Figure 3.10), with two differences. First, the function *augmentedConnectionAnalyze* invokes the augmented versions of *connectionAnalyze* functions for the constituent statements of the given control statement. Second, heap read/write sets are computed for the control statement as a whole, which is a straightforward process. As outlined in the function *controlHeapRW* in Figure 3.10, heap read/write sets for control statements are obtained by simply unioning the read/write sets of their constituent statements. With this *hierarchical* strategy, we can directly compute the dependence between any two statements, including control statements. For example, if we want to identify if a basic statement `S` can be moved above a loop statement `L`, we simply need to compare the read/write sets of statement `S` with that of statement `L`. We do not need to compare them with each individual statement inside the loop `L` to detect the potential dependence.

```
/* augmented connection analysis of a cntrl statement in SIMPLE */
fun augmentedConnectionControl(cntrlStmt, connectionSet) =
  newConnectionSet = augmentedConnectionAnalyze(cntrlStmt, connectionSet);
  [readSet,writeSet] = controlHeapRWSets(cntrlStmt);
  return [newConnectionSet,readSet,writeSet];

/* compute heap read/write sets for control stmts or for the entire function */
fun controlHeapRWSets(compStmt) =
  readSet = writeSet = {};
  foreach stmt in compStmt /* union of sets from constituent stmts */
    readSet = readSet ∪ HeapReadSet(stmt);
    writeSet = writeSet ∪ HeapWriteSet(stmt);
  return [readSet,writeSet];
```

Figure 3.10: Augmented Connection Analysis for Control Statements

3.4.4 Augmented Connection Analysis for Function Calls

The overall strategy for augmented connection analysis of function calls is outlined in the function *augmentedConnectionCall* in Figure 3.11. It involves the following individual steps:

1. **Mapping:**

   First, the connection set at the entry of the callee function is obtained by *mapping* the connection set at the call. The *map* process involves generating special symbolic names to represent pointers in caller which are out-of-scope in callee. The association between symbolic variables and the caller variables *mapped* to them, is recorded as *mapInfo*. Detailed information about the *map* process can be found in [Ghi95].

2. **Ghost Copy Assignments:**

   Next, ghost copy assignments are performed to anchor all heap-directed formal parameters and global variables at the entry of the callee function. These variables are anchored to express function-level heap read/write sets, as illustrated in section 3.3 via the parameter anchors `a@tricky_flip` and `b@tricky_flip`.

3. **Function Analysis:**

   In this step, the body of the callee function is analyzed, with the new connection set obtained after the ghost copy assignments. At the end of this analysis, we obtain the connection set valid at the exit of the function (*calleeExitCSet*).

4. **Unmapping:** Here the *calleeExitCSet* is *unmapped* to obtain the partial connection set valid after the call (*partialCallCSet*), using the map associations recorded in *mapInfo*. This set is partial because the next step, that computes heap read/write sets, can add new connection relationships to it.

5. **Read/Write Set Computation:** This is the final step, that computes heap read/write sets for the call statement, and also adds new connection relationships with respect to call-site anchor handles, in the *partialCallCSet* to obtain the complete *callCSet*.

We now illustrate the augmented connection analysis for call statements via an example based on the code fragment shown in Figure 3.12. Consider the call to function `flip` at program point `D`. At the point of the call, we have Connect(D) = {*(t@C,t), (s)*}. The connection pair {*(t@C,t)*} is generated due to the ghost copy

```
/* augmented connection analysis of callStmt, igNode: invocation graph node */
fun augmentedConnectionCall(callStmt, connectionSet, calleeFunc, igNode) =
  [calleeEntryCSet,mapInfo] = connectionMap(callStmt, connectionSet);
  postCopyCSet = funcGhostCopyAssigns(calleeFunc, calleeEntryCSet);
  /* connection set at the exit of the callee function */
  [calleeExitCSet,calleeReadSet,calleeWriteSet] =
       augmentedConnectionAnalyze(calleeFunc.body, postCopyCSet, igNode);
  calleeInfo = [calleeExitCSet,calleeReadSet,calleeWriteSet];
  /* incomplete connection set after the call */
  partialCallCSet = unmap(calleeExitCSet, mapInfo);
  /* compute read/write sets & complete the connection set valid after call */
  [callCSet,callReadSet,callWriteSet] =
       callRWSets(callStmt, partialCallCSet, calleeInfo, mapInfo);
  return [callCSet,callReadSet,callWriteSet];

/* compute heap read/write sets for the callStmt and also include the
 * connection relationships for call site anchors in the partialCallCSet */
fun callRWSets(callStmt, partialCallCSet, calleeInfo, mapInfo) =
  callReadSet = {}; callWriteSet = {}; /* initialize call read/write sets */
  callRAnchor = getAnchor(callStmt, READ);
  callWAnchor = getAnchor(callStmt, WRITE);
  /* Step 1 : focus on anchors in the caller, read/write accessed by callee */
  [callReadSet,callWriteSet] =
      callRWSetsStep1(callStmt, calleeInfo, callReadSet, callWriteSet, mapInfo);
  /* Step 2 : focus on anchors in callee that would be visible in caller */
  [callCSet,callReadSet,callWriteSet] =
      callRWSetsStep2(partialCallCSet, calleeInfo, callRAnchor,
                      callWAnchor, callReadSet, callWriteSet, mapInfo);
  return [callCSet,callReadSet,callWriteSet];

/* perform ghost copy assignments for heap-directed formals and globals */
fun funcGhostCopyAssigns(calleeFunc, connectionSet) =
  foreach heap-directed formal/global Var in calleeFunc.VarSet
    anchor = getAnchorForVar(Var, calleeFunc); /* anchor representing Var */
    copyStmt = [anchor = Var];
    connectionSet = connectionAnalyze(copyStmt, connectionSet);
  return connectionSet;
```

Figure 3.11: Augmented Connection Analysis for Call Statements

assignment `t@C = t` for the heap reference `t->x` at statement `C`. Now due to parameter passing, the call generates the following connection relationships: *(a,s)*, *(b,t)* and *(b,t@C)*. However, as the names `s` and `t` are not visible to the function `flip`, connection analysis *maps* them to special symbolic names `0+a` and `0+b`, generating the connection relationships *(a,0+a)* and *(b,0+b)*. The anchor handle `t@C` is also treated as a variable local to `main`. As it gets connected with `b`, `t@C` is also mapped to the symbolic name `0+b`. So the connection pair *(b,t@C)* gets mapped as the pair *(b,0+b)*, which is already generated. Thus the connection set at the entry to `flip` is {*(a,0+a)*, *(b,0+b)*}.

```
typedef struct point
{ double x;
  double y;
  struct point *next;
}Point;
```

```
void flip(Point *a, Point *b)
/* a gets the reflection of b */
{ Point *tmp;
  S: a->x = b->y;
  T: a->y = b->x;
}
```

```
void main()
{ Point *s, *t;
  int y;
    s = alloc_point();
    t = alloc_point();
 C: t->x = 10;
 D: flip(s, t);
 E: y = s->x;
}
```

Figure 3.12: Example for Computing Function Call Read/Write Sets

Now at the entry to function `flip`, we perform the ghost copy assignments `a@flip = a` and `b@flip = b` to *anchor* the function parameters. These assignments generate the additional connection pairs: *(a@flip,0+a)*, *(b@flip,0+b)*, *(a@flip,a)*, and *(b@flip,b)*. Also note that the symbolic names `0+a` and `0+b` actually serve as anchors inside function `flip` for the invisible variables mapped to them. This is because the process of *mapping* connects a symbolic name with all accessible variables, which are connected with the (invisible) variables mapped to it. For example `0+a` gets connected with `a`, because variable `s` mapped to it, is connected with `a`. Secondly, like an anchor handle, a symbolic variable also cannot be accessed by the program. After the initial assignment during mapping, its connection relationships can be changed only through other accessible program variables connected to it.

Considering both parameter anchors and symbolic names, we get the following connection set at the entry of function `flip`: {*(a,0+a), (b,0+b), (a@flip,0+a), (b@flip,o+b), (a@flip,a), (b@flip,b)*}. With this connection set, the heap read/write sets for the function `flip` in terms of its anchors (including symbolic anchors) will be as follows:

HeapRead(flip) = {`b@flip->y, b@flip->x, 0+b->y, 0+b->x`}
HeapWrite(flip) = {`a@flip->x, a@flip->y, 0+a->x, 0+a->y`}

Note that no other anchors are generated for this function, as the primary SSA numbers remain unchanged through the function body for variables `a` and `b`.

The heap read/write sets for the call itself are computed while *unmapping* the connection matrix from the exit of function `flip` to obtain the connection matrix valid after the call. Both, the read/write sets of the callee in the current context, and the connection relationships are used to compute them.

The sets for the function call (call to `flip` in our example) must reflect the read/write status of : (i) the anchored locations from the caller function `main` visible at the call (for example the location `t@C->x` from statement `C`), and (ii) the anchored locations from the callee function `flip`, which become visible after the call in function `main` (for example the locations `a@flip->x` and `a@flip->y`). The former set is required to identify if the call can read/write some heap location read/written by a statement *before* the call (statement `C`). The latter set is required to detect if a statement *after* the call (statement `E`) can read/write some heap location, potentially read/written by the call.

For our example program, the sets for function call `flip` should indicate that it reads the anchored location `t@C->x` from statement `C`, and that the anchored location `s@E->x` that *would-be* read at statement `E` is written by the call via `0+a->x` in HeapWrite(flip) (as they anchor the same location).

Given the sets HeapRead(flip) and HeapWrite(flip) and the connection set, *callee-ExitCSet*, valid at the exit of function `flip`, the sets at program point `D` are calculated in the following two steps, outlined in function *callRWSets* in Figure 3.11.

- **Step 1:** For an anchor handle visible at the call, the symbolic variable it is mapped to is identified, and the locations read/written with respect to the symbolic anchor are also assumed to be read/written with respect to the given anchor handle. For example, at the call to `flip`, the anchor `t@C` is visible, and is mapped to the symbolic variable `0+b`. As the locations `0+b->x` and `0+b->y` are present in the set HeapRead(flip), the locations `t@C->x` and `t@C->y` are

```
/* compute heap read/write sets for the callStmt with respect to
 * anchors from the caller, potentially read/written by the call */
fun callRWSetsStep1(callStmt, calleeInfo, callReadSet, callWriteSet, mapInfo) =
 callerAnchorSet = getAnchorSet(callStmt.callerFunc); /* caller function */
 [calleeExitCSet,calleeReadSet,calleeWriteSet] = getInfo(calleeInfo);
 foreach anchor ∈ callerAnchorSet /* each anchor in the caller */
   /* anchor is mapped to mapVar in callee */
   mapVar = getMapVar(anchor, mapInfo);
   /* set of anchors in callee connected with mapVar */
   connectedAnchorsSet = getConnectedAnchors(mapVar, calleeExitCSet);
   foreach calleeAnchor ∈ connectedAnchorsSet
     foreach anchoredLoc calleeAnchor->field ∈ calleeReadSet
       callReadSet = callReadSet ∪ anchor->field;
     foreach anchoredLoc calleeAnchor->field ∈ calleeWriteSet
       callWriteSet = callWriteSet ∪ anchor->field;
 return [callReadSet,callWriteSet];
```

Figure 3.13: Step 1 for Computing Heap Read/Write Sets for Function Calls

included in the set HeapRead(D), indicating a flow dependence from statement `C` to `D`. The overall algorithm for this step is summarized in Figure 3.13.

- **Step 2:** To represent the *would-be visible* anchored locations from the callee inside the caller, the two call-site anchor handles generated for the call-site are used. In our example, the call-site anchors are `read_anchor@D` and `write_anchor@D`, which are used to respectively represent the read and written *would-be visible* anchors. An anchor handle from the callee is considered *would-be visible* in the caller, if in the *calleeExitCSet*, it is connected with either a symbolic variable or a global variable: this indicates that variables connected with the given anchor handle will be accessible after the call, which in turn can be used to read/write the given anchored location.

  In our example program, the anchor handle `a@flip` is connected with the symbolic variable `0+a` at the exit of function `flip`. Now `0+a` will be unmapped as the variable `s` in `main`. Hence `s` inherits the connection relationships of `0+a` including those with anchor handles. However, for efficiency reasons, we assume anchor handles are local in scope to a given function. Otherwise the connection approximation of a function would need to include a large number of anchor handles. To include the connection relationships of variables with out-of-scope anchors, we represent these anchors with the two call-site anchors mentioned above.

  As already noted, in our example we use the call-site anchors `read_anchor@D` and `write_anchor@D` to respectively represent all read/written anchors. An anchor that is both read/written, would be represented by both the call-site anchors. Following this strategy the connection pairs *(0+a,a@flip)* and *(0+b,b@flip)* are respectively unmapped as the pairs *(s,write_anchor@D)* and *(t,read_anchor@D)*. The sets HeapRead(flip) and HeapWrite(flip) are also similarly unmapped. Including the contribution from Step 1, we get the following sets for program point `D`:

  HeapRead(D) = {`read_anchor@D->y, read_anchor@D->x, t@C->y, t@C->x`}
  HeapWrite(D) = {`write_anchor@D->x, write_anchor@D->y`}

  Now at the statement `E: y = s->x`, the connection pair *(s,write_anchor@D)* is valid. So the ghost copy assignment `s@E = s`, will generate the connection pair *(s@E,write_anchor@D)*. Thus the anchor `write_anchor@D` will be included in the set HeapRead(E), indicating a flow dependence from statement `D` to statement `E`, as required. The overall algorithm for this step is summarized in Figure 3.14.

```
/* unmap heap read/write sets for anchors in callee that would be
 * visible in the caller, in terms of the call site anchors. Also
 * connect the call site anchors with the variables that make the
 * callee anchors visible in the caller */
fun callRWSetsStep2(callCSet, calleeInfo, callRAnchor,
                    callWAnchor, callReadSet, callWriteSet, mapInfo) =
 [calleeExitCSet,calleeReadSet,calleeWriteSet] = getInfo(calleeInfo);
 foreach anchor ∈ calleeAnchorSet /* each anchor in the callee */
   foreach pair  (anchor, Var) ∈ calleeExitCSet
     if (isSymbolic(Var) || isGlobal(Var)) /* Var is visible in the caller */
       /* caller vars mapped to Var: a global is mapped to itself */
       mappedVarSet = getMappedVars(Var, mapInfo);
       foreach mappedVar in mappedVarSet
         readFlag = writeFlag = 0;
         foreach anchoredLoc anchor->field ∈ calleeReadSet
           callReadSet = callReadSet ∪ callRAnchor->field;
           readFlag = 1;
         foreach anchoredLoc anchor->field ∈ calleeWriteSet
           callWriteSet = callWriteSet ∪ callWAnchor->field;
           writeFlag = 1;
         if (readFlag == 1) /* connect mappedVar with read anchor */
           callCSet =  (callRAnchor, mappedVar) ∪ callCSet;
         if (writeFlag == 1) /* connect mappedVar with write anchor */
           callCSet =  (callWAnchor, mappedVar) ∪ callCSet;
  return [callCSet,callReadSet,callWriteSet];
```

Figure 3.14: Step 2 for Computing Heap Read/Write Sets for Function Calls

Program	Description	Principal Data Structures
alvinn	SPEC92 C benchmark	Static Arrays
water	Molecular Dynamics	Arrays and Lists
health	Health Care Simulation	Quad Tree and Lists
graphics	Various Graphics Routines	Non-recursive Heap Structures
circuit	Sparse Matrix Solver	Doubly-linked Lists
nbody	N-body simulation	Octree Data Structure
specear	SPEC92 C benchmark	Static/Dynamic Arrays
em3d	Electromagnetic Simulation	Dynamic Arrays and Lists
yacr2	Channel Routiner	Arrays of Heap Structures
ks	Graph Partitioning	Dynamic Arrays and Lists
vor	Voronoi Diagrams	Splay Tree and Lists
power	Power System Optimization	k-ary Tree
nrcode2	Vector C benchmark	Dynamic Arrays
blocks2	Computational Biology	Dynamic Arrays and Lists
sim	Computational Biology	Dynamic Arrays and Lists
eigen	Eigenvalue Computation	Dynamic Arrays

Table 3.2: Benchmark Descriptions

3.5 Empirical Evaluation of the Analyses

We have evaluated the efficiency of our analyses with respect to a set of 16 pointer-intensive C benchmark programs, drawn from the SPEC92, SPLASH-2 [WOT+95], Olden [RCRH95], Irvine [HHN94] and Wisconsin [ABS94] benchmark suites. A brief description of the benchmarks is provided in Table 3.2. The table also summarizes the principal data structures used by the benchmarks.

In Table 3.3 we provide the analysis times measured on an *UltraSparc* machine in *seconds*. The points-to and connection timings respectively include the additional time spent in computing stack and heap read/write sets. The table also provides the following information for each program: (i) number of SIMPLE statements, (ii) number of stack and heap related indirect references, (iii) the average number of anchor handles generated per function, and (iv) the average number of anchor handles generated per indirect reference.

An interesting observation is that our context-sensitive pointer analyses are quite efficient for our benchmark set. We have achieved this efficiency via two optimizations,

Program	#Stmts	Stack Refs	Heap Refs	Anchors/ Func	Anchors/ Ref	Points-to Time	Connection Time
alvinn	340	42	0	0	0.00	0.19	0.00
water	2829	31	569	16	0.53	6.90	7.90
health	392	4	116	5	0.55	0.51	0.66
graphics	1161	21	250	8	0.51	4.21	0.86
circuit	1679	10	465	8	0.51	0.99	2.24
nbody	1110	54	177	3	0.78	1.93	1.92
specear	3176	143	146	2	0.34	2.77	1.20
em3d	279	3	51	5	0.65	0.13	0.09
yacr2	3064	10	665	9	0.50	7.27	13.31
ks	652	44	90	4	0.46	0.59	0.58
vor	1623	63	479	8	0.48	2.87	2.59
power	741	29	151	3	0.09	1.05	0.15
nrcode2	312	0	53	10	0.19	0.16	0.05
blocks2	1149	82	279	9	0.48	1.66	4.30
sim	1845	69	340	6	0.28	1.56	2.60
eigen	296	11	44	2	0.30	0.10	0.05

Table 3.3: Analysis Statistics

namely *function exclusion* and *memoization*, which significantly reduce the interprocedural overhead of the context-sensitive analyses [GH96a]. The function exclusion optimization identifies functions in the program, that do not update heap-directed pointers and also do not involve accesses to heap locations. This information is obtained from points-to analysis information and stack read/write sets. Subsequently, these functions are excluded from the search space of connection analysis, enabling it to analyze a sparser program.

The *memoization* optimization was initially proposed in [Ema93]. The goal of this optimization is to reduce the number of times a function is analyzed. Under context-sensitive schemes, a function is re-analyzed for each of its invocation contexts. With memoization, the input/output analysis information for each visit to the function is recorded. During a subsequent visit to the function, the analysis first checks if the current input information matches a previously seen input for the function. If a match is found, the corresponding output information is directly used, without actually analyzing the function body. Opportunities for memoization arise due to the fact that even though pointer targets may be different for different invocation contexts, pointer relationships such as connection relationships tend to remain similar.

The above observations also indicate that for larger programs, if interesting sections of the program can be identified using linear points-to analyses [Ste96, SH97b], precise information for these sections can be obtained efficiently via context-sensitive approaches. We are presently experimenting with this approach to efficiently analyze larger programs.

Another observation from the data in Table 3.3, is that the average number of anchor handles generated per indirect reference is about 0.50 for most of the benchmarks. Thus by using SSA numbers, we reduce the number of anchor handles needed by almost 50%. Further note that the number of anchor handles generated per function is less than 10 for all the benchmarks (except 16 for *water*). This indicates that our strategy of using SSA numbers and generating special handles for call-sites, effectively restricts the number of anchor handles introduced.

3.6 Understanding Anchor Handles

In sections 3.3 and 3.4, we illustrated how to compute heap read/write sets with respect to anchor handles. In this section, we review the properties of anchor handles and formally show that anchor-based heap read/write sets provide *conservatively correct* information for dependence detection.

Intuitively, the main property of an anchor handle, say `a@S`, is that after the ghost copy assignment `a@S = a`, it *continues* to point to the data structure pointed to by `a` at statement `S`, throughout the program. The reason behind is that a stack-resident pointer can be made to point to a different data structure only by updating the pointer itself. This is possible to do for program variables like pointer `a`. However, anchor handles are symbolic names not accessible to the program, and cannot be updated after the relevant ghost copy assignment.

For example, in Figure 3.6 the anchor `a@S` continues to point to the same node (data structure) throughout the function, unlike `a`. Thus at any statement (executing after statement `S`[2]), if a pointer can point to this data structure, it should be connected with the anchor `a@S`. In the example program, `tmp` points to this data structure due to the statement `tmp = a`, and hence gets connected with `a@S`. Next, the statement `b = tmp` makes `b` point to the same data structure and connects it with `a@S`.

Let us now formally show that anchor-based heap read/write sets are *conservatively correct* i.e. they *always* indicate a dependence if it actually exists, but may also indicate a dependence in some cases when it does not exist. Consider the statements `S` and `T`, belonging to some function `foo`, below:

```
S: a->x = rhs;
   ..
   ..
T: lhs = b->x;
```

We want to prove the following theorem.

Theorem 3.6.1 *If there exists an execution path P in the program, along which an instance of statement* `T`*, namely T_i, follows an instance of statement* `S`*, namely S_i, and the heap references* `a->x` *at S_i and* `b->x` *at T_i access the same heap location, then connection analysis should report* (a@S,b@T) $\in$ *Connect(T).*

Informally, we want to prove that, when the above assumptions hold, connection analysis would report anchor handle `a@S` to be connected with the anchor `b@T` at statement `T`. This would signify that `a@S->x` is possibly the same location as `b@T->x`, and indicate the flow dependence from `S` to `T` as required. Also note that by an execution path P in the theorem, we do not mean a static control path between two statements, but an actual run time trace of the program.

[2]This property is required to ensure that the anchor has been introduced in connection analysis via the ghost copy assignment `a@S = a`.

Proof: Let the data structure pointed to by `a` at S_i be denoted as D. Let Y be the set of pointers that point to D at S_i. Clearly, $\forall\ y \in Y$, *(a,y)* $\in$ Connect(S). This is because connection analysis guarantees that if *at a given program point*, two pointers can point to the same data structure, they are reported to be connected. Further *(a,y)* $\in$ Connect(S) $\Rightarrow$ *(a@S,y)* $\in$ Connect(S), since connection analysis performs the ghost copy assignment `a@S = a` at `S`.

Let Z be the set of pointers that are made to point to the data structure D due to some statement instance along P between S_i and T_i. The set Z is empty at S_i. We now follow the construction of set Z along P. Our main goal is to prove the following lemma.

Lemma 3.6.1 *If a statement instance, M_i, along P makes a pointer,* `ptr`*, fall into the set Z, then the corresponding statement* `M` *in the program creates the connection pair (ptr,a@S).*

We prove the lemma by induction on the number of elements added to the set Z. We first prove it for the *first* element added to Z, then prove it for the *nth* element added to Z, assuming it to be true for the previous *(n - 1)* elements added.

Basic Step: Let M_i be the *first* statement instance along P, that follows S_i, and makes a pointer, say `newPtr`, point to D (i.e. fall into the set Z). To make `newPtr` point to D, M_i must assign to it a pointer, say `currPtr`, that *currently* points to D. At M_i, `currPtr` must be from the set Y (as the set Z is as yet empty), and further `currPtr` should not have been *updated* between S_i and M_i (i.e. still point to D).

Let M_i be an instance of the statement `M` in the program. To make `newPtr` point to D at M_i, statement `M` must be a basic heap statement (Figure 3.3(a)) of one of the following forms: (i) `newPtr = currPtr`, (ii) `newPtr = currPtr->f`, (iii) `newPtr = &(currPtr->f)`, (iv) `newPtr = currPtr op k`, or (v) `newPtr->f = currPtr`. According to connection analysis rules summarized in Table 3.1, all these statements make `newPtr` (and all pointers connected with `newPtr` for statement (v)), connected with all pointers *currently* connected with *currPtr*.

Since `currPtr` $\in$ Y, we also have *(currPtr,a@S)* $\in$ Connect(S). Further, since `currPtr` is not updated between S_i and M_i along P, there must be a control path, say C, in the program from `S` to `M` along which `currPtr` is not updated. Now a connection pair can be killed only when one of the two pointers involved is updated. This is because the kill sets for the statements `p->f = q` and `p->f = NULL` are empty

(Table 3.1), and the data flow merge operator for two connection matrices is a simple logical OR, which ensures that no connection pairs are killed at control flow merge points. Thus the connection pair (*currPtr,a@S*) from statement `S` must get propagated to statement `M` along the control path C, as along C neither `currPtr` nor `a@S` are updated (anchor handles cannot be updated after the relevant ghost copy assignment).

So we should have *(currPtr,a@S)* $\in$ Connect(M). As discussed above, statement `M` should connect `newPtr` with all pointers connected with `currPtr`. Hence it should generate the connection pair *(newPtr,a@S)*, as required. Thus for the *first* pointer that falls into the set Z, Lemma 3.6.1 holds.

Inductive Step: Let N_i be the statement instance that adds the *nth* pointer, `nPtr`, to set Z. Similar to the argument for the basic step, `nPtr` should be assigned some pointer, `tPtr`, that currently points to the data structure D, through one of the basic heap statements. Now we can have two cases for the pointer `tPtr`: (i) it points to D at S_i i.e. it belongs to Y and has not been updated since, or (ii) it is made to point to D at some intermediate statement instance L_i, between S_i and N_i (i.e. put into set Z), and has not been updated since.

The first case is similar to the case for basic step, and using the same argument we can deduce that *(tPtr,a@S)* $\in$ Connect(N) and that statement `N` generates the connection pair *(newPtr,a@S)*. For the second case, `tPtr` is the *kth* element put into set Z (by statement instance L_i), where $k < n$. So according to induction hypothesis, the connection pair *(tPtr,a@S)* is generated by the corresponding statement `L` in the program. Since `tPtr` is not updated between L_i and N_i, using the same argument as in the basic step, we can deduce that the connection pair *(tPtr,a@S)* is propagated from statement `L` to statement `N`, and hence statement `N` generates the pair *(nPtr,a@S)*.

Thus we have proved the induction hypothesis for the *nth* element falling into the set Z, assuming that it holds for the previous *(n - 1)* elements. This proves Lemma 3.6.1.

Now we get back to the proof for Theorem 3.6.1. We first prove that *(b,a@S)* $\in$ Connect(T), by using similar arguments as for proving *(tPtr,a@S)* $\in$ Connect(N) during the inductive step of Lemma 3.6.1. According to our initial assumption, pointer `b` at T_i points to the same heap object as pointer `a` at S_i. Thus pointer `b` points to the data structure D at T_i. We can have the same two cases for `b` at T_i as we had for the pointer `tPtr` while proving the inductive step: (i) it points to D at S_i i.e. it belongs to the set Y and has not been updated since, or (ii) it is made to point to D

at some intermediate statement instance L_i, between S_i and T_i (i.e. put into set Z), and has not been updated since.

Again, the first case is similar to the case for the basic step of Lemma 3.6.1, and using the same argument we can deduce that *(b,a@S)* $\in$ Connect(T). For the second case, suppose `b` is last put into set Z by statement instance L_i. So according to Lemma 3.6.1, the connection pair *(b,a@S)* is generated by the corresponding statement `L` in the program. Since `b` is not updated between L_i and T_i, using the same argument as in the basic step, we can deduce that the connection pair *(b,a@S)* is propagated from statement `L` to statement `T`. In either case, the pair *(b,a@S)* is propagated to statement `T`. Subsequently, the ghost copy assignment, `b@T = b` at `T` will generate the connection pair *(a@S,b@T)*, and we would have *(a@S,b@T)* $\in$ Connect(T), as required. QED.

3.7 Summary

In this chapter we briefly reviewed connection analysis, described its augmentation with anchor handles and discussed the computation of heap read/write sets with respect to the anchor handles. We demonstrated the efficiency of our analyses on a set of 16 C benchmark programs. Finally, we also proved the correctness of the anchor-based heap read/write sets. In the next chapter we evaluate the effectiveness of our read/write set information, in the context of a variety of compiler applications.

Chapter 4

Applications

In the last chapter, we described techniques for accurate computation of read/write sets in the presence of pointers. The goal of this chapter is to evaluate the impact of our sharper read/write information on improving the effectiveness of a range of compiler applications. We have used our implementation of stack and heap read/write set analyses as the basis for extending several standard scalar compiler optimizations like loop invariant removal and common subexpression elimination, and a new optimization called location invariant removal. We have also defined how to use this information to guide several more advanced compiler applications including array dependence testing, detecting heap structures that are run-time constants, and as a guide to prefetching. We have also built a program-understanding tool, that displays the read/write information to the user via a Web browser.

The rest of this chapter is organized as follows. First in section 4.1, we describe the scalar optimizations we have used in our experiment, and in section 4.2, present a detailed empirical evaluation with respect to a set of 12 pointer-intensive C benchmark programs. In section 4.4, we discuss the use of our read/write sets for array dependence testing and present relevant empirical data for 6 array-based C programs. In section 4.5, we describe our Web browser-based tool to display read/write information, and in section 4.6, we briefly sketch how the read/write information can be used for detecting run-time constants, for prefetching in the context of recursive data structures, and for identifying memory leaks. Finally, in section 4.7 we discuss related work on using the results of pointer analysis.

4.1 Scalar Optimizations

To study the impact of our read/write set information on scalar optimizations, we have implemented two standard scalar optimizations in our compiler framework, namely, loop invariant removal (LIR) and common subexpression elimination (CSE). These optimizations were chosen because of their wide applicability in general and for pointer references in particular. Additionally, they are implemented in most of the optimizing compilers, which makes it easier to compare our results with state-of-the-art technology.

The LIR and CSE optimizations are illustrated in Figure 4.1. These optimizations are called *scalar* optimizations, because traditionally they are applied to computations involving scalar variables. For example, for the code segment shown in in Figure 4.1(a), LIR optimization is applied to the expression `x * y`, and CSE optimization is applied to the expression `x * i`. Both the expressions perform a computation involving scalar variables. Figure 4.1(b) shows the optimized code, which saves two multiplications per iteration as compared to the original code.

In our implementation, we have *extended* the scope of these *scalar* optimizations to include *pointer-based references*. For example, in Figure 4.1(c), the pointer reference `q->i` is loop-invariant. Our version of LIR optimization replaces this pointer reference with a scalar temporary, as shown in Figure 4.1(d). The scalar replacement does not save any computation. However, it saves one memory reference per iteration, as the scalar temporary `tq` can be register-allocated. With the increasing memory-processor performance gap in modern computer systems [HP96], reduction in memory traffic is of critical importance, and can result in substantial performance gains.

Besides the LIR and CSE optimizations, we have also implemented a new optimization, which we call *location invariant removal* (LcIR). It is a variation on the LIR optimization and is specifically applicable to pointer references.

Thus, all three optimizations are designed to eliminate redundant computations and reduce memory accesses in the given program. They have been implemented at the SIMPLE intermediate representation. After performing the optimizations, the transformed SIMPLE AST is either dumped as an optimized C program giving a *source-to-source* transformation, or is directly given as input to the EARTH-McCAT backend for the EARTH-MANNA multithreaded architecture, as shown in Figure 2.1 (page number 27).

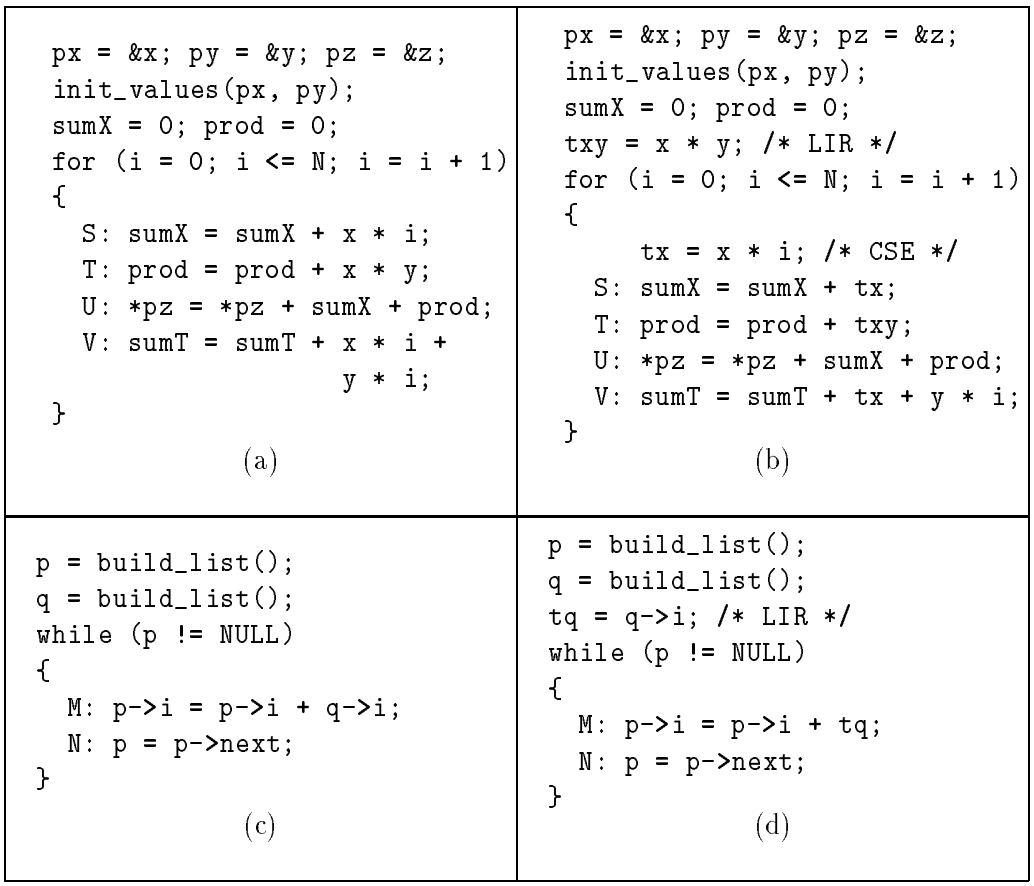


Figure 4.1: Illustrating Scalar Optimizations

4.1.1 Loop Invariant Removal

The goal of this optimization is to detect computations which are invariant over all iterations of the given loop, and move them outside the loop. We have illustrated this optimization in Figure 4.1(b). In Figure 4.1(d), we have also demonstrated its applicability to replace indirect pointer references inside loops with scalar temporaries that can be register-allocated. Loop invariant pointer references frequently arise in C programs, typically in deeply embedded heap accesses, loop condition tests and inner loops.

We show some examples from our benchmark programs (in *simplified* version) in Figure 4.2. The label `L` indicates the program point from where the loop-invariant has been moved. Parts (a) and (b) show loops from the Olden [RCRH95] benchmarks *health* and *power* respectively. From both the loops, the compiler pulls out an address calculation required to access arrays embedded inside heap objects. Parts (c) and (d) show loops from the benchmark *circuit* [HHN94]. In part (c) the loop test involves a loop invariant which is pulled out. In part (d) the access `(* botRH).row` is loop-invariant, but the given statement cannot be moved out of the loop as the left hand side is also a pointer access. But we are still able to substitute the loop-invariant pointer access with a scalar access.

We now discuss in detail our implementation of the LIR optimization. Our implementation attempts to detect loop-invariant computations, address calculations, and *read* memory accesses and move them out of the loop. In a *simplified* C program (Figure 2.2), a computation can be of the form: (i) lhs = op scalar, or (ii) lhs = scalar_1 op scalar_2; and an address calculation can be of the form: (i) lhs = `&a`, (ii) lhs = `&(a[i])`, or (iii) lhs = `&(a->field)`; where lhs can be a scalar, an array reference or an indirect reference, and the term scalar on rhs also represents constants.

Further a read memory access includes any right-hand-side pointer dereference (possible forms: (i) `*a`, (ii) `a[i]`, and (iii) `a->field`), simple structure or array reference (`a.b`, `a[i]`), and any *read* access to an address-exposed or a global variable. Basically, all variable references that cannot be register-allocated without accurate alias information, are considered as memory accesses. We have already explained why address-exposed variables cannot be register-allocated. Global variables also fall into this category because they can be updated by any function call. To safely promote them to registers, accurate read/write sets for function calls are required, for which in turn pointer analysis information is needed.

We now describe how we use our read/write sets for detecting loop invariants.

```
/* loop invariant access */
temp_18 = (* village).forward;
for (i = 0; i <= 3; i = i + 1)
{ L:
  temp_17 = temp_18[i];
  temp_16 = get_tot_hosps(temp_17);
  number = (number + temp_16);
}
```

(a) health

```
/* loop invariant access */
temp_2 = (* t).feeders;
for (i = 0; i <= 9; i = i + 1)
{ temp_1 = (i * 10);
  l = build_lateral(temp_1,20);
  L:
  temp_2[i] = l;
}
```

(b) power

```
temp_336 = (* rightCH).col;
/* loop invariant access */
temp_337 = (* realPivotE).col;
while (temp_336 != temp_337)
{ rightCH = (* rightCH).ncolH;
  temp_336 = (* rightCH).col;
  L:
}
```

(c) circuit

```
if (botE != 0) /* guard */
{
  /* loop invariant access */
  invar_temp_42 = (* botRH).row;
}
while ( (botE != 0) )
{ L:
  (* botE).row = invar_temp_42;
  botE = (* botE).ncolE;
}
```

(d) circuit

Figure 4.2: Loop Invariants

Consider a SIMPLE statement `S: lhs = rhs` inside a loop `L`. To determine if `rhs` is loop-invariant, we proceed in the following steps:

- **Stack Comparison:** We first compare the stack read set of statement `S` (Read(S)) with the stack write set of the loop `L` (Write(L)), by computing the intersection set Sset = Read(S) ∩ Write(L). We can have the following three cases: (i) Sset is empty, (ii) Sset only contains the symbolic location *heap*, and (iii) Sset is non-empty, but case (ii) does not apply. For case (i), `rhs` is marked as loop-invariant. For case (iii), `rhs` is marked as *loop variant*, as some location(s) read by `rhs` may be written by some statement in the loop. For case (ii), a further comparison of heap read/write sets is performed as discussed below.
- **Heap Comparison:** Here we compute the intersection set Hset = HeapRead(S) ∩ HeapWrite(L). If Hset is empty, `rhs` is marked as loop-invariant, else it is marked as loop-variant.

Note that if the `lhs` of statement `S` includes an indirect reference, it can also contribute to the stack read set Read(S). For example, if `lhs` is of the form `(*p).item`, `p` is included in Read(S). We exclude such locations from the set Read(S), when using it for the above comparisons, because we only want to compare the locations read by `rhs`.

If `rhs` is detected as loop-invariant, and it can be *safely* moved out of the loop (discussed below), we modify the statement as `S: lhs = invar_temp` and insert the statement `invar_temp = rhs` just before the loop. Alternatively the statement itself can be moved out if the following additional constraints are satisfied [ASU88]: (i) `lhs` is a scalar variable, say `x`, (ii) this variable `x` is not assigned by any other statement in the loop, (iii) no use of `x` in the loop is reached by a definition of `x` other than at `S`, and (iv) definition of `x` at `S` does not reach any use of `x` outside the loop. These conditions by design hold for the temporary variables generated during the *simplify* phase of our McCAT compiler. So in our implementation we are able to move the statement itself out of the loop in majority of the cases, as opposed to generating additional temporaries of the form `invar_temp`.

When `rhs` involves a pointer dereference or a divide operation, we add two extra checks. First, loop-invariant removal is performed only if the statement `S` is

not embedded inside any conditional statement inside the loop, and is not preceded by a `break`, `continue` or `return` statement. Second, the new statement `invar_temp = rhs` outside the loop is guarded by the loop condition. The first condition guarantees (for our compositional SIMPLE representation) that the dereference/divide will definitely be executed if control enters the loop, and the second condition ensures that the dereference/divide is performed out of the loop, only when control will enter the loop. In our implementation, if we pull any dereference/divide out of a loop, we put all invariant statements placed out of that loop as well as the loop body inside an `if` statement guarded by the loop condition (see example in Figure 4.2(d)). Thus we introduce at most one additional branch per loop. The overall algorithm for loop invariant removal is summarized in Figure 4.3.

4.1.2 Location Invariant Removal

We now describe our new optimization, we call *location invariant removal* (LcIR). Any memory reference (i.e. a pointer reference, or an access to a global or address exposed variable) that accesses the same memory location in all iterations of a loop is considered to be *location invariant*. A location invariant access can also be replaced by a scalar access, under some constraints. This optimization is similar in spirit to the scalar replacement technique proposed for array references [CCK90]. Consider the loop in Figure 4.4(a). In this loop the heap access `r->i` is location invariant as the origin pointer `r` is not written inside the loop. We can replace all accesses to `r->i` with a scalar, say `temp_0`, by initializing `temp_0` as `r->i` and then assigning it to `r->i` outside the loop (Figure 4.4(b)). This would save two memory accesses per iteration, as the scalar `temp_0` can be register-allocated. In Figure 4.8(d) (page number 93), we show an example from our benchmark program *alvinn*, where LcIR optimization is applied.

The conditions for a pointer access to be location invariant are: (i) the origin pointer is not written inside the loop, and (ii) if it is an array access, the array indices are not written inside the loop. These checks are not needed for the case of global and address exposed variables. However, to actually perform location invariant removal, we need to perform further tests. Consider again the loop in Figure 4.4(a). If the heap reference `p->i` in some iteration refers to the same location as the reference `r->i`, the scalar replacement as shown in Figure 4.4(b) cannot be done. In this iteration, a wrong value will be read for the reference `p->i`, as all writes to the read location are being diverted to the variable `temp_0`.

```
/* determine if stmt contains a loop-invariant reference/computation */
fun loopInvar(stmt, loopStmt) =
  sSet = stackCompare(stmt, loopStmt, rhs);
  if (sSet  == EMPTY) /* no conflicts */
    return INVARIANT;
  if (sSet == ONLYHEAP) /* only heap access conflict */
    hSet = heapCompare(stmt, loopStmt, rhs);
    if (hSet  == EMPTY)
      return INVARIANT;
  return VARIANT;

/* check if stack read/write sets conflict */
fun stackCompare(stmt, loopStmt, side) =
  readSet = Read(stmt, side); /* read set only for stmt rhs */
  writeSet = Write(loopStmt); /* write set for the entire loop */
  sSet = intersectSets(readSet, writeSet);
  return (sSet);

/* check if heap read/write sets conflict */
fun heapCompare(stmt, loopStmt, side) =
  hReadSet = HeapRead(stmt, side); /* heap read set only for stmt rhs */
  hWriteSet = HeapWrite(loopStmt); /* heap write set for the entire loop */
  hSet = intersectSets(hReadSet, hWriteSet);
  return (hSet);

/* move the invariant out of the loop */
fun invarCodeMotion(stmt, loopStmt) =
  lhsRef = getRef(stmt, lhs); rhsRef = getRef(stmt, rhs);
  if (goodScalar(lhsRef)) /* lhsRef is scalar that satisfies all constraints */
    moveStmt = stmt;
  else /* lhsRef involves a bad scalar or a component/array/indirect ref */
    invarTemp = getTemp(rhsRef);
    moveStmt = buildStmt(invarTemp, rhsRef); /* invarTemp = rhsRef */
    substitute(stmt, rhsRef, invarTemp); /* substitute rhsRef with invarTemp */
  insertBefore(moveStmt, loopStmt); /* insert moveStmt before the loop */
  if (derefOrDivide(moveStmt)) /* contains a divide or a pointer dereference */
   /* guard the moveStmt and loopStmt with the loop condition */
   guardWithLoopCond(moveStmt, loopStmt);
```

Figure 4.3: Removing Loop Invariants

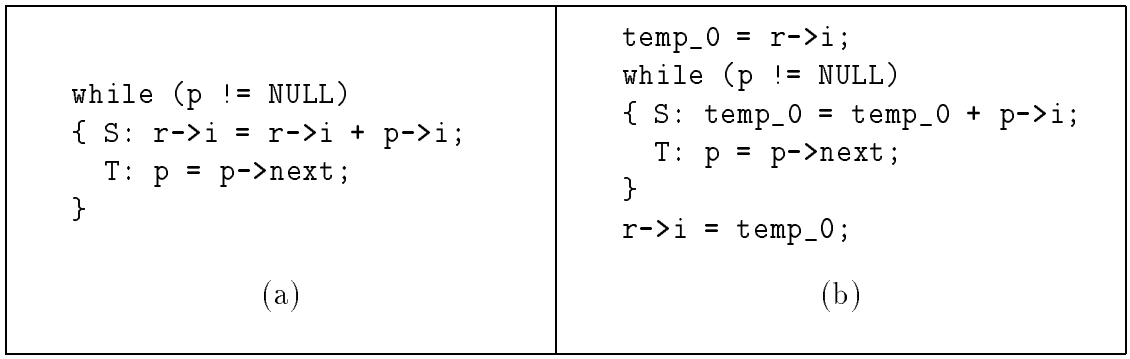


Figure 4.4: Location Invariants

Consider a location invariant access `r->i` at statement `S` which refers to the location *S_loc*. To *safely* perform location invariant removal, the following additional checks are required. First, we need to identify all memory references in the loop that can access this location. We use our stack and heap read/write sets to collect this information. Next, we need to ensure that all these pointer references are location invariant and they also always access the location *S_loc*. This is a much harder problem. In our analysis we simply check that all these references are syntactically equivalent to the given location invariant reference. If this is not the case, we do not apply LcIR for the given reference. Finally, when moving a pointer reference for location invariant removal, we make the same extra checks as described for loop invariant removal. The algorithm for location invariant removal is summarized in Figure 4.5.

4.1.3 Common Subexpression Elimination

The goal of this optimization is to avoid recomputation of an expression, whose value has been previously computed and has not changed since. We had illustrated this optimization in Figure 4.1(b) at the beginning of the chapter (page number 79). This is a widely applicable optimization, and is implemented in most of the optimizing compilers. The CSE optimization also proves to be very effective in the pointer context. In programs using recursive pointer data structures, memory accesses involving several dereferences like `r->f->g->arr[k]` are not uncommon. Suppose this array is accessed several times in a loop. The programmer would use a macro like `ARRAY(r,k)` to avoid typing in the whole sequence of dereferences at every access. However, in the absence of alias information, the compiler has to re-compute the entire pointer expression for every access, even if only the array index `k` is varying. Further

```
/* identify if the stmt contains a location invariant reference */
fun locInvar(stmt, loopStmt) =
  locRef = getRef(stmt, lhs); /* reference to be substituted with a scalar */
  locVar = getBaseVar(locRef); /* Base var for locRef */
  if (locVar ∉ Write(loopStmt)) /* locRef is location invariant */
    refStmtList = getRefStmtList(locRef, loopStmt);
  if (refStmtList == NULL)
    /* cannot identify all stmts that exactly access same location as locRef */
    return NULL;
  locTemp = getTemp(stmt, lhs); /* get a temp variable for substitution */
  beforeStmt = buildStmt(locTemp, locRef); /* locTemp = locRef */
  afterStmt = buildStmt(locRef, locTemp); /* locRef = locTemp  */
  insertBefore(beforeStmt, loopStmt); /* insert beforeStmt before the loop */
  insertAfter(afterStmt, loopStmt); /* insert afterStmt just below the loop */
  if (hasDeref(locRef)) /* contains a pointer dereference */
   /* guard all three stmts with the loop condition */
   guardWithLoopCond(beforeStmt, loopStmt, afterStmt);
  foreach refStmt in refStmtList /* stmts that use locRef */
    substitue(refStmt, locRef, locTemp); /* replace locRef with locTemp */

/* get the list of stmts in the loop accessing the location invariant locRef */
fun getRefStmtList(locRef, loopStmt) =
  list = NULL;
  foreach stmt in body(loopStmt)
    if (readOrWrite(locRef, stmt)) /* accesses the same location as locRef */
      if (!exactRef(locRef, stmt)) /* reference is syntactically not locRef */
        return NULL; /* LcIR cannot be performed for locRef */
      addToList(list, stmt); /* add stmt to list of stmts that use locRef */
  return list;
```

Figure 4.5: Removing Location Invariants

examples showing application of CSE optimization to pointer references, taken from our benchmark programs are shown in Figure 4.9 (page number 94).

We have implemented the common subexpression elimination(CSE) algorithm from [ASU88]. It is global in scope, i.e., is applied over a function, and is based on first collecting *available expressions*. An expression `x + y` is considered *available* at a program point `P`, if it is computed along all paths to `P` and after the last computation of `x + y` on a given path, both `x` and `y` are not updated. If an expression is available at a statement `S`, and if `S` uses this expression, a CSE transformation is applied to replace the expression with a temporary that holds the value of the expression from its last computation (illustrated in Figure 4.1(b)). The algorithm for CSE is summarized in Figure 4.6.

```
/* perform CSE for the stmt if it contains an available expression */
fun doCSE(stmt) =
  availList = getAvailList(stmt); /* get list of avail exprs at the stmt */
  rhsExpr = getRef(stmt, rhs); /* get the expr on the rhs of the stmt */
  matchList = matchingStmts(rhsExpr, availList); /* stmts that use rhsExpr */
  if (matchList != NULL)
    cseTemp = getTemp(rhsExpr);
    foreach stmtM ∈ matchList
      cseStmt = buildStmt(cseTemp, rhsExpr); /* cseTemp = rhsExpr */
      insertAfter(cseStmt, stmtM); /* insert cseStmt after stmtM */
  substitute(stmt, rhsExpr, cseTemp); /* substitute rhsExpr with cseTemp */
```

Figure 4.6: Eliminating Common Subexpressions

In our implementation, while collecting available expressions, we target the same three kinds of expressions as we consider for loop-invariant removal, i.e., computations, address calculations and *read* memory accesses. The overall algorithm for collecting available expressions is summarized in Figure 4.7. It is straightforward to identify the expressions made *available* by a statement. To accurately identify expressions made *unavailable* by a statement, we use our read/write set information.

In the function *killExpr*, for each available expression, we first compare the stack read set of the statement that computes it (*stmtL*), with the stack write set of the statement under consideration (*stmt*). If the sets conflict, the expression becomes unavailable after the statement. For example, if the available expression is `(*p).item`,

```
/* compute expressions available after the stmt */
fun availExprs(stmt, availList) =
  storeList(stmt, availList); /* list of expr's available at the stmt */
  genCSE(stmt, availList); /* add expr on rhs of the stmt to availList */
  killCSE(stmt, availList); /* find expr's made unavailable by the stmt */

/* identify expressions made available by the stmt */
fun genExpr(stmt, availList) =
  rhsExpr = getRef(stmt, rhs); /* get the expr on the rhs of the stmt */
  if (isOfInterest(rhsExpr)) /* computation/memory ref/address calculation */
    if (!isAvailable(rhsExpr, availList)) /* not already available */
      addToList(availList, stmt); /* add stmt to available expressions list */

/* identify expressions made unavailable by the stmt */
fun killExpr(stmt, availList) =
  foreach stmtL ∈ availList
    sSet = stackCompare(stmtL, stmt, rhs);
      if (sSet  != EMPTY) /* stack accesses conflict */
        if (sSet == ONLYHEAP) /* conflict is only due to heap accesses */
          hSet = heapCompare(stmtL, stmt, rhs);
           if (hSet  != EMPTY) /* heap access conflict */
             /* expr on rhs of stmtL becomes unavailable */
             deleteFromList(availList, stmtL);
        else  /* conflict is not only due to heap accesses */
          /* expr on rhs of stmtL becomes unavailable */
          deleteFromList(availList, stmtL);

/* given the two available expression lists, identify expressions
 * that remain available after a control flow merge point */
fun mergeAvailLists(availListI, availListJ) =
  foreach stmtI ∈ availListI
    rhsExprI = getRef(stmtI, rhs);
    matchList = matchingStmts(rhsExprI, availListJ);
    if (matchList != NULL) /* some stmts in availListJ use rhsExprI */
      addToList(stmtI, matchList); /* list of stmts that use rhsExprI */
      addToList(matchList, newAvailList); /* rhsExprI remains available */
  return newAvailList;
```

Figure 4.7: Computing Available Expressions

and *stmt* writes to the variable `p`, the stack read/write sets will conflict and the expression will become unavailable.

However, if the stack sets conflict only due to a common access to the symbolic variable *heap*, we compare the heap read set of *stmtL* with the heap write set of *stmt*. In case of a conflict, the expression becomes unavailable. For example, for the available expression `(*p).item`, if *stmt* does not write to `p`, but writes to `(*p).item` itself, it will become unavailable after *stmt*.

Finally, since we perform global CSE, at control flow merge points we need to *merge* the lists of expressions available from the different control paths. The function *mergeAvailLists* summarizes this operation. The basic idea is that an expression is considered available after the merge point, if it is available along all control paths reaching that point. When more than two control paths reach a merge point (e.g. for `switch` statements), the operation is applied iteratively, considering two paths at a time.

4.2 Experimental Results for Scalar Optimizations

In this section, we study the experimental results obtained by applying the above three optimizations using our read/write sets to a suite of 12 pointer intensive C benchmarks programs. The first twelve entries in Table 3.2 (page number 70), briefly describe these benchmarks and the principal data structures they use. Other characteristics of the benchmark programs like the number of statements, and the number of indirect references, are presented in Table 3.3 (page number 71).

We have collected both static (compile-time) and dynamic (runtime) statistics for the above benchmark suite. First, we present the compile-time data, which indicate the applicability of a given optimization for a program, when using both the stack and heap read/write sets (section 4.2.1). We then examine how many of these optimization opportunities are due to the presence of heap read/write sets (section 4.2.2). Finally, we provide runtime measurements to measure the ultimate benefit of the stack and heap read/write sets for decreasing the number of memory accesses, decreasing the number of instructions executed, and decreasing the running time of optimized programs (section 4.2.3).

4.2.1 Optimizations Applied with Stack and Heap Read/Write Sets

Our first experimental results report how many opportunities arise for our scalar optimizations, when run with both the stack and heap read/write sets available. The purpose of this data is to demonstrate the types of optimizations possible, and to show that a significant number of opportunities arise in pointer-intensive benchmarks.

In Table 4.1, the three multicolumns respectively give the number of times the three optimizations: loop invariant removal (Loop Invars), location invariant removal (Loc Invars) and common subexpression elimination (CSE), are applied for a program. Each multicolumn is further divided into several labeled columns, which give the data for different types of expressions. The denotations for labels are as follows: (i) gl: global variables, (ii) ae: address exposed variables, (iii) addr: address calculations of the form `&(a[i])` or `&(a->field)`, (iv) ind: indirect references (`*a`, `a->field`, `a[i]` where `a` is a pointer), simple array references (`a[i]` where `a` is not a pointer), and simple component references (`a.field`), and (v) expr: expressions involving computation (`lhs = op scalar`, `lhs = scalar_1 op scalar_2`). Note that location invariant optimization is not applicable for expressions involving address calculations or computation. Simple component references are put in the "ind" category because they are typically not treated as scalars by compilers, and are not typically register-allocated.

From the data in Table 4.1, we make the following important observations:

- For several benchmarks globals and address exposed variables contribute significantly to the number of loop invariants (*water*, *specear* and *yacr2*). All of these benchmarks use a number of global variables which are initialized only once in the program (at the beginning), so are invariant for most of the loops. Our loop-invariant algorithm replaces them with temporaries which can be register allocated. Figure 4.8(a) shows an example from the *specear* benchmark where the global variable `Earlength` is substituted with the temporary `temp_invar_19`.

- A significant number of loop-invariant address calculations are found for the benchmarks *water*, *health*, *graphics*, *nbody* and *yacr2*. They typically arise due to accesses to arrays embedded insided (heap allocated) structures. An example from the *nbody* benchmark is shown in Figure 4.8(b).

Program	Loop Invars					Loc Invars			CSE				
	gl	ae	addr	ind	expr	gl	ae	ind	gl	ae	addr	ind	expr
alvinn	12	0	6	3	0	0	0	3	0	0	2	6	5
water	449	26	104	33	265	0	1	10	0	4	111	143	202
health	1	0	14	2	2	0	0	3	0	0	1	19	0
graphics	0	0	25	8	6	0	0	0	0	0	11	93	14
circuit	0	4	0	53	22	0	0	0	0	1	0	66	27
nbody	4	2	44	9	11	2	0	0	5	0	32	23	9
specear	25	1	2	9	55	13	0	0	0	0	2	62	86
em3d	0	0	0	9	2	0	0	0	0	0	0	2	2
yacr2	107	7	94	111	21	7	0	2	47	0	106	149	116
ks	1	5	12	19	10	1	0	7	0	0	17	26	10
vor	0	1	1	0	0	2	0	0	15	0	43	247	22
power	0	0	4	2	0	0	0	8	0	0	0	62	9

Table 4.1: Distribution of Optimizations Applied

- Invariant indirect references (*ind* column) are found in all the benchmarks (being pointer intensive). They typically arise in nested loops, where the outer loop is traversing the nodes of a recursive data structure, and the inner loop operates on the node itself. Indirect references with respect to the node in the inner loop provide opportunities for invariant detection. An example from *em3d* is shown in Figure 4.8(c) where we find three indirect references with respect to the pointer `nodelist`, invariant for the inner loop.

  In summary, we found indirect invariants of the following forms in the inner loops (assume `node` is the current node traversed by the outer loop):

  1. `temp = &(*node).array; lhs = temp[1]`, where `temp` points to a heap-embedded array (*water*).
  2. `temp = (*node).pointer`, where the field `pointer` points to a heap-allocated array and `temp` is used to access its elements (*yacr2*, *em3d*).
  3. `while (i < (*node).count))`, (*circuit*) and `lhs = temp[(*node).index]` (*ks*), where the scalar fields are invariant.

- A significant number of invariants are also found in the *expr* category. For *water*, the majority of the *expr* invariants are of the form `lhs = op scalar`, where `op`

is a type cast operation and `scalar` represents the address of a heap-embedded array. For other benchmarks, majority of these invariants are of the form `lhs = scalar_1 op scalar_2`, where the two scalar variables involved are typically global variables or constants.

- We find limited applications of location invariant removal in our benchmarks. However, they prove to be critical in the dynamic context. Opportunities for location invariant removal arise mainly due to summation/reduction operations. Typical examples include : (i) storing up the sum of all node values in the header node, and (ii) an inner loop storing the sum of its node values, to the current outer loop node. Figure 4.8(d) shows a loop from the *alvinn* benchmark where location invariant optimization is applied.

- CSE optimization finds numerous applications in all our benchmarks, particularly for address calculations and indirect references. To give further insight, we show representative example applications of CSE from different benchmarks in Figure 4.9. Part(a) shows a loop from *water* where repeated array address calculations are eliminated. Part(b) shows a loop from *health*. Here the loop pointer is `list`, however each loop iteration accesses fields of the heap object pointed to by the pointer `(*list).patient`. With CSE optimization, we compute this pointer only once for each iteration, as opposed to once for every field access (as in the original program). Part(c) shows a loop from the benchmark *circuit*. Here the pointer expression `(*ch).ncolH` is common for the loop test and the statement advancing the loop pointer `ch`. With CSE optimization we compute the expression only once per iteration, as opposed to twice. Finally part(d) shows a code fragment from a loop in *vor* where a pointer expression calculated for parameter passing is reused later via CSE as the function call does not modify its value.

4.2.2 Benefits of Using Heap Read/Write Sets

The data shown in Table 4.1 is for the case when we use both the points-to based stack read/write sets, and the connection-based heap read/write sets. In order to measure the additional benefits achieved due to heap read/write sets, we also applied the above optimizations with only stack read/write sets. We compare the data for the two cases in Table 4.2 for LIR, Table 4.3 for LcIR and Table 4.4 for CSE optimization.

```
/* LIR */
temp_invar_18 = (*input);
/* LIR */
temp_invar_19 = EarLength;
for (i = 0; i < temp_invar_19;
                      i = i + 1)
{ InputState[i] = temp_invar_18;
}
```

(a) specear

```
/* LIR */
 temp_79 = (*p).un.cell.subp;
 for (k = 0; k <= 7; k = k + 1)
 { temp_78 = temp_79[k];
   if (temp_78 != 0)
   { temp_82 = (dsq / 4.0);
     walksub(temp_78, temp_82);
   }
 }
```

(b) nbody

```
while (nodelist != 0)
{ /* LIR */
  temp_18 = (*nodelist).from_nodes;
  /* LIR */
  temp_19 = (*nodelist).coeffs;
  /* LIR */
  temp_17 = (*nodelist).from_count;
  for (i = 0; i < temp_17;  )
  { other_node = temp_18[i];
    coeff = temp_19[i];
    value = (*other_node).value;
    temp_21 = (*nodelist).value;
    temp_22 = (coeff * value);
    (*nodelist).value = (temp_21 -
                         temp_22);
    i = i + 1;
  }
  nodelist = (*nodelist).next
}
```

(c) em3d

```
/* guard */
if (sender <= end_sender)
{
  /* LcIR */
  temp_invar_2 = (*receiver);
  for ( ; sender <= end_sender; )
  { temp_31 = temp_invar_2;
    temp_34 = sender;
    sender = (sender + 1);
    temp_33 = (*temp_34);
    temp_36 = weight;
    weight = (weight + 1);
    temp_35 = (*temp_36);
    temp_32 = (temp_33 * temp_35);
    temp_invar_2 = (temp_31 +
                    temp_32);
  }
  (*receiver) = temp_invar_2;
} /* end guard */
```

(d) alvinn

Figure 4.8: Examples for Loop Invariants (LIR) and Location Invariants (LcIR)

```
while (curr_box != 0)
{ temp_14 = &(*curr_box).coord;
  i = temp_14[0];
  temp_15 = temp_14; /* CSE */
  j = temp_15[1];
  temp_16 = temp_14; /* CSE */
  k = temp_16[2];
  ... /* statements deleted */
  curr_box =
         (*curr_box).next_box;
}
```

(a) water

```
while (list != 0)
{ temp_43 = (*list).patient;
  temp_42 = (*temp_43).time_left;
  (*temp_43).time_left =
                   (temp_42 - 1);
  temp_45 = temp_43; /* CSE */
  temp_44 = (*temp_45).time_left;
  if ((temp_44 == 0))
  { ... /* statements deleted */
    temp_57 = temp_43; /* CSE */
    (*temp_57).time_left = 10;
    temp_58 = temp_43; /* CSE */
    temp_59 = (*temp_58).time;
    ... /* statements deleted */
  }
  list = (*list).forward;
}
```

(b) health

```
temp_234 = (*ch).ncolH;
temp_cse_2 = temp_234;
while ( (temp_234 != 0) )
{ temp_235 = (*ch).col;
  mapUnk[temp_235] =
         (*ch).whichUnknown;
  ch = temp_cse_2; /* CSE */
  temp_234 = (*ch).ncolH;
  temp_cse_2 = temp_234;
}
```

(c) circuit

```
temp_62 = (*act).v1;
temp_63 = (*n).v1;
temp_61 = point_equal(
          temp_62, temp_63);
if ((temp_61 != 0))
{ p = (*act).v2;
}
else
{ p = temp_62; /* CSE */
}
```

(d) vor

Figure 4.9: Common Subexpression Elimination (CSE) Examples

The columns labeled as S and H respectively give the numbers for the case with only stack read/write sets being used, and the case with both stack and heap read/write sets being used. The main observations from these tables are as follows:

Program	gl		ae		addr		ind		expr		Total	
	S	H	S	H	S	H	S	H	S	H	S	H
alvinn	12	12	0	0	6	6	3	3	0	0	21	21
water	449	449	26	26	**92**	**104**	**7**	**33**	**241**	**265**	**815**	**877**
health	1	1	0	0	**13**	**14**	**0**	**2**	2	2	**16**	**19**
graphics	0	0	0	0	25	25	**2**	**8**	6	6	**33**	**39**
circuit	0	0	4	4	0	0	**32**	**53**	22	22	**58**	**79**
nbody	4	4	2	2	44	44	**8**	**9**	11	11	**69**	**70**
specear	25	25	1	1	2	2	**7**	**9**	**54**	**55**	**89**	**92**
em3d	0	0	0	0	0	0	**2**	**9**	2	2	**4**	**11**
yacr2	107	107	7	7	94	94	**20**	**111**	21	21	**249**	**340**
ks	1	1	5	5	12	12	19	19	10	10	47	47
vor	0	0	1	1	1	1	0	0	0	0	2	2
power	0	0	0	0	4	4	**0**	**2**	0	0	**4**	**6**

Table 4.2: Comparing Loop Invariants Detected

- The total number of loop invariants and/or common subexpressions eliminated increases moderately with heap read/write sets, for all the benchmarks (except for *alvinn* that does not have any heap references). We see the most substantial increase in the indirect references (*ind*) category. For example, the number of loop invariants for *yacr2* increases from 20 to 111, and the number of CSE optimizations for *water* increases from 115 to 143, in the *ind* category. This happens because in our benchmark set, the majority of the indirect references access the heap, and sharper heap read/write sets expose more opportunities for optimizations.

- Improved information about indirect references also leads to a moderate increase in the *expr* category. For example, if the variable `i` is defined in terms of an indirect reference `i = (*p).item`, which is identified as loop-invariant, the expression `x + i` can also be identified as loop-invariant. Similarly if `i` is used in an address expression `t = &(a[i])`, the address expression can also be identified as loop-invariant. In the *expr* category for the *water* benchmark,

Program	gl		ae		ind		Total	
	S	H	S	H	S	H	S	H
alvinn	0	0	0	0	3	3	3	3
water	0	0	1	1	**0**	**10**	**1**	**11**
health	0	0	0	0	**0**	**3**	**0**	**3**
graphics	0	0	0	0	0	0	0	0
circuit	0	0	0	0	0	0	0	0
nbody	2	2	0	0	0	0	2	2
specear	13	13	0	0	0	0	13	13
em3d	0	0	0	0	0	0	0	0
yacr2	7	7	0	0	2	2	9	9
ks	1	1	0	0	7	7	8	8
vor	2	2	0	0	0	0	2	2
power	0	0	0	0	8	8	8	8

Table 4.3: Comparing Location Invariants Detected

Program	gl		ae		addr		ind		expr		Total	
	S	H	S	H	S	H	S	H	S	H	S	H
alvinn	0	0	0	0	2	2	6	6	5	5	13	13
water	0	0	4	4	**100**	**111**	**115**	**143**	**189**	**202**	**408**	**460**
health	0	0	0	0	1	1	**6**	**19**	0	0	**7**	**20**
graphics	0	0	0	0	11	11	**37**	**93**	14	14	**62**	**118**
circuit	0	0	1	1	0	0	**43**	**66**	27	27	**71**	**94**
nbody	5	5	0	0	32	32	23	23	9	9	69	69
specear	0	0	0	0	2	2	**53**	**62**	**85**	**86**	**140**	**150**
em3d	0	0	0	0	0	0	**1**	**2**	2	2	**3**	**4**
yacr2	47	47	0	0	**100**	**106**	**120**	**149**	116	116	**383**	**418**
ks	0	0	0	0	17	17	**25**	**26**	10	10	**52**	**53**
vor	15	15	0	0	**39**	**43**	**219**	**247**	22	22	**295**	**327**
power	0	0	0	0	0	0	**49**	**62**	**5**	**9**	**54**	**71**

Table 4.4: Comparing CSE Optimizations Applied

we see an increase in the number of invariants from 241 to 265, and in CSE optimizations from 189 to 202.

- In most of the benchmarks, stack read/write sets are able to detect majority of the optimization opportunities. This happens because of two reasons. First, for the case of globals and address exposed variables, heap read/write information does not bring any added advantage. Second, if a loop or code fragment does not involve any write to *heap*, all heap-related invariants/common subexpressions can still be detected for it without a heap analysis.

- The number of location invariants increases only for *water* and *health* as these are the only benchmarks in which heap-related location invariant expressions arise, along with *ks*. However, the two heap-based location invariants from *ks* arise in loops which do not involve any other write access to *heap*, so they can already be detected without needing heap read/write sets.

As noted above, only with stack read/write sets that conservatively estimate the heap, heap-related invariants/common subexpressions may be detected in several cases. Likewise, it is also possible that an optimizing compiler that conservatively handles pointer references, could also detect many of the optimization opportunities that our analyses detect. For example, the CSE transformations shown in Figure 4.9(a) can also be applied without any pointer analysis information (in fact gcc applies them).

4.2.3 Runtime Improvements with Stack and Heap Read/Write Sets

In order to measure the additional benefits of our analyses over a state-of-the-art optimizing compiler, we have compared our results with the GNU C compiler [Sta92] (gcc version 2.7.2) working at the highest level of optimization (with -O3 flag). Since our transformations are source-to-source and are performed at the SIMPLE intermediate representation, we performed the following experiment. We produced three sources for each benchmark program: (i) the dump of the SIMPLE representation of the program (*plain* version), (ii) the dump of the SIMPLE representation of the program *after* the above three optimizations are applied with only stack read/write sets (*Sopt* version), and (iii) the optimized dump with both stack and heap read/write sets being

used (*Hopt* version). The SIMPLE dumps are just *simplified* C programs which can be compiled by any native C compiler.

Next, we compile all three versions with the gcc compiler with *-O3* optimization flag, and compare the *run-time* performance of the *opt* versions over the *plain* version. Now any difference in the performance can be solely attributed to our transformations. The overall setting for our experimental framework is shown in Figure 4.10.

We have collected the following run-time statistics:

- The total number of memory references made during program execution. This is an important metric as the main effect of applying the above optimizations on pointer expressions, globals and address exposed variables, should be the reduction of memory references.
- The total number of instructions executed. This reflects how many instructions could be eliminated due to loop invariant removal. Also these source-to-source optimizations can enable the compiler to be less conservative and produce better code, which can lead to a reduction in number of instructions.
- The run time of the program measured using the "/usr/bin/time" utility on an *UltraSparc* machine with only single user logged on. The run time was calculated as the sum of the system and user time reported by the "time" utility. Also the run time was averaged over three runs of the program.

We collected the first two statistics using the EEL based [LS95] QPT2 tool developed by Jim Larus, which instruments the program executable to give exact counts. However, note that run time reported is *not* from the QPT2-instrumented versions of the executables.

The comparison of the above statistics is presented in Table 4.5. The three multicolumns labeled "Mem Refs", "Insns" and "Run Time" respectively give the data regarding the number of memory references made, number of instructions executed and the run time. The columns labeled "Sopt" and "Hopt" in the multicolumns labeled "%Decrease" respectively give the percentage decrease achieved in number of memory references or instructions executed, by the *Sopt* and *Hopt* versions over the *plain* version. The columns labeled "Abs Decr" give the actual decrease in the number of memory references/instructions (in *millions*) achieved by the *Hopt* version

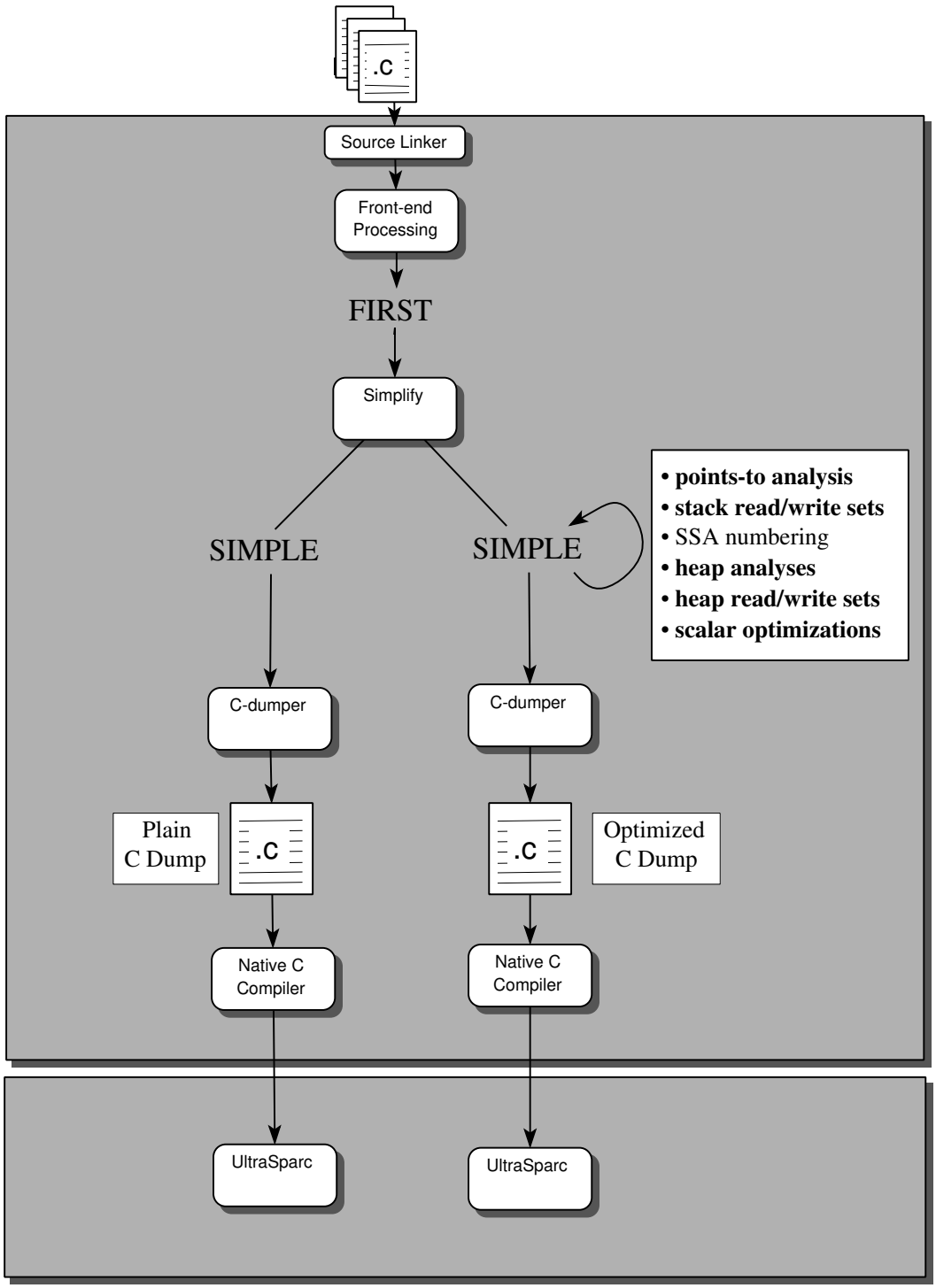


Figure 4.10: The Framework for Evaluating the Benefits of Scalar Optimizations

over the *plain* version. Finally, in the "Run Time" multicolumn, the first column (labeled "Base Time") gives the run time in *seconds* for the *plain* version. The next two columns respectively show the percentage speedup obtained by the *Sopt* and *Hopt* versions over the *plain* version.

Program	Mem Refs			Insns			Run Time		
	%Decrease		Abs	%Decrease		Abs	Base	%Speedup	
	Sopt	Hopt	Decr	Sopt	Hopt	Decr	Time	Sopt	Hopt
alvinn	35.56	35.56	684.92	11.64	11.64	682.04	42.70	10.30	10.30
water	15.59	15.64	388.16	4.50	4.51	407.13	64.10	5.77	6.08
health	0.00	15.41	135.27	-0.35	5.00	122.01	139.30	-1.87	1.36
graphics	12.41	14.32	236.30	7.87	9.08	236.10	42.40	1.89	3.07
circuit	10.01	13.39	67.22	1.59	1.63	34.87	54.30	0.92	1.47
nbody	8.13	8.14	81.25	-0.19	-0.18	-6.00	37.30	0.54	0.54
specear	7.49	7.49	293.27	5.64	5.64	817.16	107.00	1.78	2.90
em3d	0.83	3.30	7.95	0.08	0.23	5.87	16.40	1.22	1.22
yacr2	1.91	2.85	1.33	2.10	2.20	11.69	32.90	3.34	4.26
ks	2.01	2.01	36.64	0.71	0.71	37.09	43.90	0.91	0.91
vor	0.17	0.93	9.50	0.02	0.13	5.51	78.20	6.65	8.31
power	-0.01	-0.17	-0.41	-0.01	-0.05	-0.44	12.70	0.00	0.00

Table 4.5: Dynamic Improvements over *gcc -O3*

The main observations from this tables are as follows:

- The optimized versions achieve a significant reduction in the number of memory references. The highest is 35.56% for *alvinn*, while six other benchmarks achieve greater than 7% reduction. For *alvinn*, the main factor proves to be the location invariant removal shown in Figure 4.8(d) that applies to three critical inner loops. For *specear*, the pointer-based array reference `state[i+1]` arises twice (on rhs) in its critical loop in function `agc`. Between the two references there is a write via another pointer-based array reference `output[i]`. Without pointer information gcc is not able to apply CSE across this write, while we can, and it brings most of the reduction.

  For other benchmarks, invariants and common subexpressions spread all across the program contribute. Finally, for the *power* benchmark we actually see an

increase in the number of memory references, despite the numerous applications of all optimizations (Table 4.1). This happens because in this benchmark some pointer expressions remain invariant through a function and are used all across it. Via CSE, all but the first occurrence of this expression are substituted with a temporary. Such temporaries end up having long lifetimes, causing the register allocator to introduce spills, and perform worse than original. Clearly, a compiler that applies CSE optimization keeping in view the consequent increase in register pressure, can avoid this situation.

The above observations highlight the applicability of our optimizations to pointer expressions in particular. They also indicate that there may not always be a direct correlation between the number of times optimizations are applied to run time improvements.

- For five benchmarks, from 4-11% reduction is achieved in the number of instructions executed. Again, for some benchmarks we see an increase instead. This happens due to pulling out invariant expressions which would not be computed for all iterations if left inside the loop.

- The percentage decrease figures are always equal or higher for the *Hopt* version compared to the *Sopt* version, with the difference being most marked for *health* and significant for *graphics*, *circuit* and *em3d*. All these benchmarks use recursive heap data structures, so heap read/write sets bring added benefits.

- We see run time speedup of 10.30% for *alvinn*, 8.31% for *vor*, 6.08% for *water* and 4.26% for *yacr2*. These speedup figures are quite significant in the context of our scalar optimizations. Further they are achieved over "gcc -O3". While reduction in memory references and instructions executed, always translates into a speedup, the speedup obtained is not always in direct proportion. For example, for *yacr2* and *vor*, the percentage decrease figures are much less than for *health* or *circuit*. The reason they obtain better speedup is that our source-to-source transformations sometimes enable the native C compiler to perform better instruction scheduling due to substitution of pointer references with scalars (which applies for these benchmarks). For the same reason, even with the same instruction and memory reference counts, the *Hopt* version for *specear* achieves better speedup than the *Sopt* version.

4.3 Performance Improvement for EARTH-C Programs

The effects of our sharper read/write sets and the scalar optimizations presented above, have also been studied in the context of parallel EARTH-C [HTZ+96] programs run on the EARTH-MANNA multithreaded architecture [HMT+95], as shown in Figure 4.11. The detailed results are presented in [TGHG97]. Here we briefly discuss the main points.

The EARTH-MANNA architecture emulator runs on top of the MANNA parallel machine [BGSP94], which consists of multiple nodes connected by an interconnection network. It supports a *distributed shared memory* model. Each node (consisting of two processors) has its own local memory, and the aggregate of the local memories of all the nodes represents a global memory address space. Thus an access to the memory of another processor (remote access) is performed via the interconnection network and is considerably more expensive than an access to the local memory.

The experiments were performed on EARTH-C programs for four applications that primarily use recursive heap data structures (taken from the Olden suite). These programs contain a significant number of pointer references, which typically involve remote memory accesses. When LIR, LcIR and CSE optimizations are applied to such references, they are substituted with scalar temporaries, and thus remote memory accesses are replaced with local accesses. This results in substantial reduction in the number of remote memory operations performed at run time: up to 98% for the *mst* benchmark that computes a minimum spanning tree, and 32% for the *em3d* benchmark.

Our sharper read/write information also enables the EARTH-C compiler to build more accurate data dependence graphs (DDGs). With more accurate DDGs, the compiler is able to find more parallelism, schedule remote accesses earlier to hide latency, and generate longer threads which reduces the number of threads generated and the synchronization cost associated with thread-switching.

As a result of the above optimizations, significant additional run time speed-ups are obtained: up to 25% for *em3d*, 8% for *perimeter* (computes the perimeter of a quad-tree encoded raster image), and 7% for *mst*. Note that the speed-ups would be even higher for distributed memory systems like networks of workstations (NOWs), where remote memory accesses are significantly more expensive.

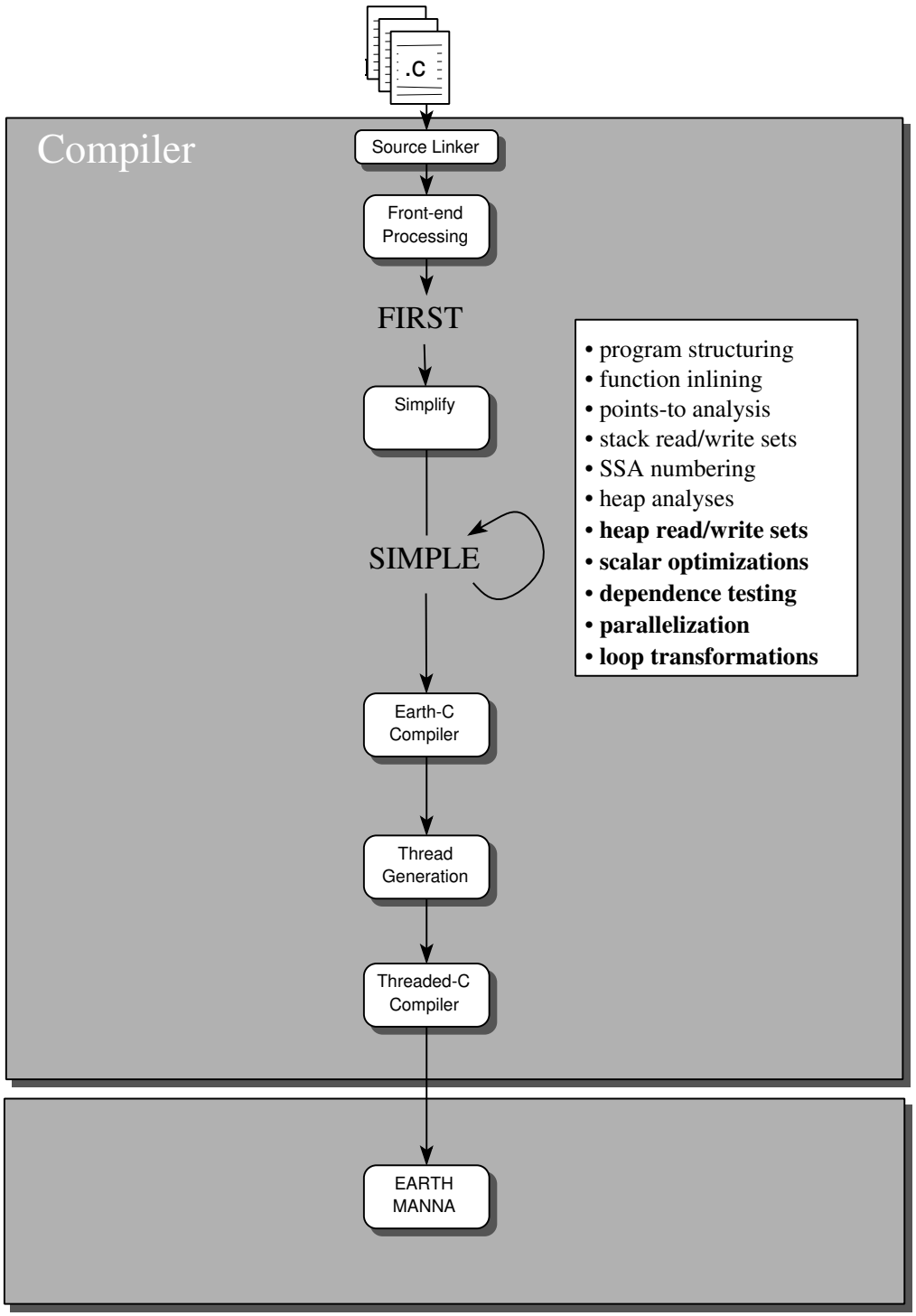


Figure 4.11: Using Scalar Optimizations for Optimizing EARTH-C Programs

4.4 Improving Array Dependence Tests

Scientific applications written in C also use arrays as principal data structures. However, unlike FORTRAN, these arrays are mostly implemented using pointers to dynamically-allocated storage. Further even statically-allocated arrays are often passed as pointer parameters. The pointer-based array references pose new problems for the array dependence tester (ADT). Consider the following simple example loop:

```
for (i = 1; i < 100; i++)
{ S: a[i] = a[i] + 1;
  T: c[i] = b[i-1] + a[i];
}
```

If the variables `a`, `b` and `c` are static integer arrays (declared as `int a[100], b[100], c[100]`), the ADT can easily identify that the loop has a flow dependence from `S` to `T`, but no loop-carried dependences. However, if these variables are declared as integer pointers which point to dynamically-allocated storage or to statically-allocated arrays, the situation becomes more complex. Now, the ADT cannot assume that two *syntactically* different array references are always independent. For example, if `a` and `b` point to the same heap object/static array, we have a loop-carried flow dependence from `S` to `T`. Using our pointer analyses, we can easily check against such possibilities. For the above loop, if either `a`, `b` and `c` point to different static arrays, or if the anchor handles `a@S`, `c@T`, and `b@T` are not connected with each other, the situation becomes identical to the static case.

We have measured the effectiveness of our pointer analyses for more precise ADT, using a set of 6 array-based C programs (described in Table 3.2, page number 70). We used the ADT implemented in the McCAT C compiler (McADT) [Jus94, Lap97] for our study. On encountering pointer-based array references, McADT invokes a routine called *checkAliasToSameVar*, that checks if the two pointer-based arrays references actually access the same array. This routine initially used the information from points-to analysis to identify if the two given pointers can point to a common stack location. We modified this routine to additionally use the information from anchor-augmented connection analysis. Now, if the only common stack location for the two pointers is the symbolic location *heap*, we further check if the anchors representing the pointers are connected.

Using McADT, we collected the following ADT statistics for each benchmark: (i) the number of array pairs tested, (ii) the number of dependences detected, and (iii) the number of *forall* loops found using the ADT results. Clearly, one would like to eliminate as many dependence tests as possible, since each test is potentially expensive, and spurious tests may lead to spurious dependences. Reducing the number of dependences is beneficial both for better fine-grain parallelism, and for exposing more *forall* loops. More *forall* loops lead to more coarse-grain parallelism.

The data is shown in Table 4.6. The columns labeled Plain and Heap respectively show the numbers for ADT without any pointer information, and with both points-to and connection information. One can see a significant reduction in both array pairs tested and dependences detected. We are also able to find more *forall* loops for *specear*, *nrcode2*, *blocks2* and *alvinn*. These results indicate that pointer analyses can make ADT considerably more effective. In fact, some commercial compilers like *pgcc* (from Portland Group Inc.) provide pragmas to get similar information (like *-Msafeptr* for the user to indicate that certain pointers do not share storage with other pointers/arrays). Finally, for the other three benchmarks, the array dependences disproved, fall into loops which are actually not *forall* loops. So we do not see an increase in the number of *forall* loops for them.

Program	Array Pairs		Dependences		Forall	
	Plain	Heap	Plain	Heap	Plain	Heap
specear	200	68	138	46	9	19
nrcode2	70	7	70	7	3	6
blocks2	16	13	10	7	6	9
alvinn	20	8	20	8	6	7
yacr2	28	7	28	7	23	23
sim	77	7	77	7	3	3
eigen	15	5	15	5	2	2

Table 4.6: Results for Array Dependence Analysis

4.5 Program Understanding/Debugging

Our summary read/write information can also be used as a program understanding/debugging aid. For example, consider a procedure `foo(List *a, Tree *b)`, with the following summary information:

HeapWrite(foo) = { `a@foo->index, b@foo->left, b@foo->right` }
HeapRead(foo) = { `a@foo->index, a@foo->next, b@foo->left, b@foo->right` }

Based on this information we can make interesting observations about the effect of function `foo` on the data structures passed to it. The absence of `a@foo->next` in the HeapWrite set indicates that the function does not affect the structure of the list (does not add/delete nodes), and that it only modifies the scalar field `index` of some or all the nodes in the list (`a@foo->index` $\in$ HeapWrite(foo)).

On the contrary, the presence of `b@foo->left` and `b@foo->right` in the HeapWrite set indicates that the `left` and/or `right` pointer fields are modified for some nodes of the tree. This could imply that either new nodes have been added to the tree, or some nodes have been deleted, or some nodes have simply swapped their children. According to our experience with benchmarks, the useful information is again the negative information: which fields are not updated by the given function, with information about the *link* pointer fields being specially useful. Also, due to the hierarchical nature of our read/write sets, such information can be obtained with respect to other program constructs like loops, conditionals and function calls.

To nicely display such information to the user, we have developed a tool that uses a Web browser, as shown in Figure 4.12. To this end, we modified our C-dump utility to produce HTML version of the given program, with each statement decorated with a hyperlink to a CGI script passing the *unique* statement ID as a hidden parameter. We also produced compressed files containing the pointer analysis, and read/write sets information for each statement along with its ID. When the user clicks on a statement or a function, the CGI script (written in perl) is invoked with the statement/function ID as its argument. It looks for this ID in the current *infofile* and displays the information associated with it in a user friendly form in the right frame.

We show three snapshots of the browser showing the read/write information in Figures 4.13, 4.14 and 4.15. The picture in Figure 4.13 shows a snapshot for the

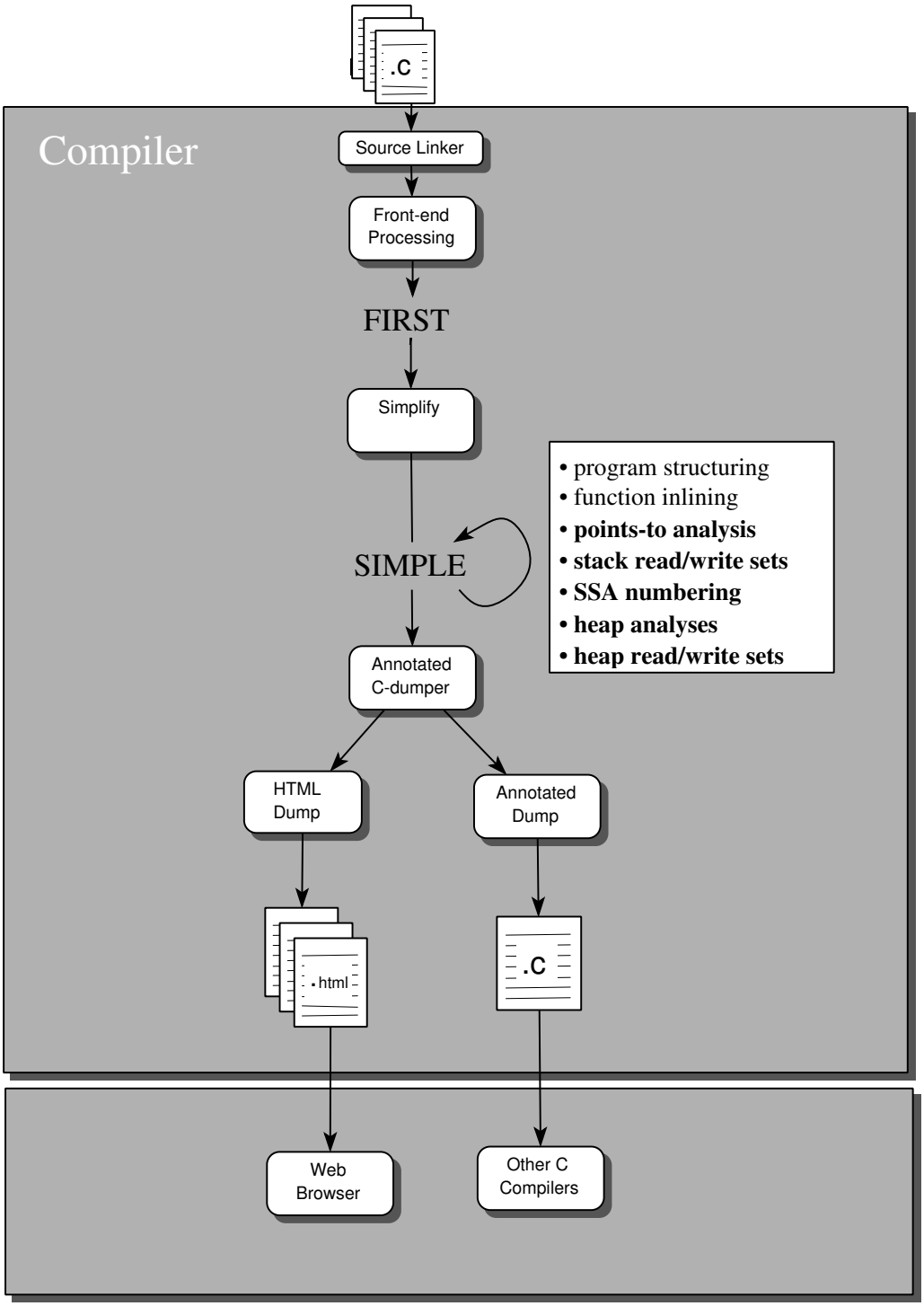


Figure 4.12: Using Read/Write Sets for Program Understanding and Other Applications

example program shown in Figure 1.5 (page number 22). The top frame in the picture shows the types of information that can be viewed by clicking at the given link. The information displayed by each link is as follows:

- *Show Intro*: The introduction document to the Web browser-based program understanding tool.
- *Points-to*: The set of locations pointed to by a dereferenced pointer at a given statement.
- *Connection*: The set of pointers connected at a statement involving heap indirection.
- *Stack R/W Sets*: Points-to based stack read/write sets.
- *Heap R/W Sets*: Connection-based heap read/write sets.
- *Field-Sets*: The set of fields accessed with respect to appropriate anchors.

The right frame in the picture shows the stack read/write sets for the call to function `sum` from inside `main`. This information was obtained by first clicking at the link *Stack R/W Sets* in the top frame, and then clicking at the statement containing the call.

The second picture shown in Figure 4.14 displays the heap read/write sets for the function `tricky_flip` shown in Figure 3.6 (page number 55). The sets are shown in terms of anchor handles and the fields they access. As discussed in section 3.4, in our implementation, an anchor handle is simply a program variable subscripted with its current SSA number. Also in the right frame, each field is shown with the data type it belongs to and clicking on the type displays its definition.

The source program in the left frame also shows each variable subscripted with its current SSA number. SSA numbers are not displayed when an address operation is applied on a variable as its value is not being used. In the heap read/write sets, a variable subscripted with SSA number zero represents its anchor at the entry of the given function. The sets shown in the right frame indicate that the field `x` is read with respect to the anchor $\mathtt{b}_2$, as well as the anchors $\mathtt{a}_0$ and $\mathtt{a}_1$. Similarly the write sets indicate that the field `x` is written with respect to the anchors $\mathtt{a}_0$ and $\mathtt{a}_1$. From

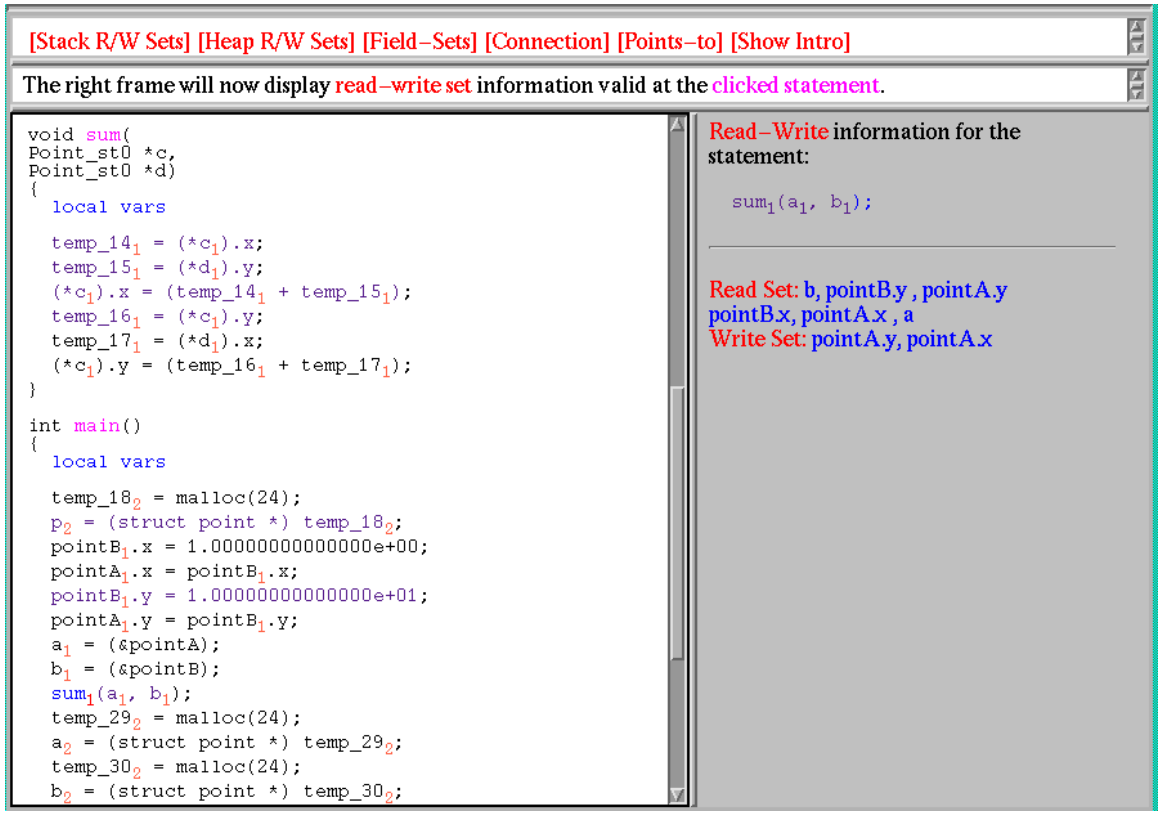


Figure 4.13: Browser Snapshot showing Stack R/W sets

this information, the flow dependence from the first statement in `tricky_flip` to its last statement can be easily deduced.

In the source program on the left frame, each call to a user-defined function is also shown subscripted with a call expression number (Figure 4.13): the *nth* call-site in a function gets the call expression number n. Each heap related call-site (with call expression number n), has two anchor handles associated with it: cs_read$_n$ and cs_write$_n$. The two anchor handles summarize the heap locations read/written by the function call.

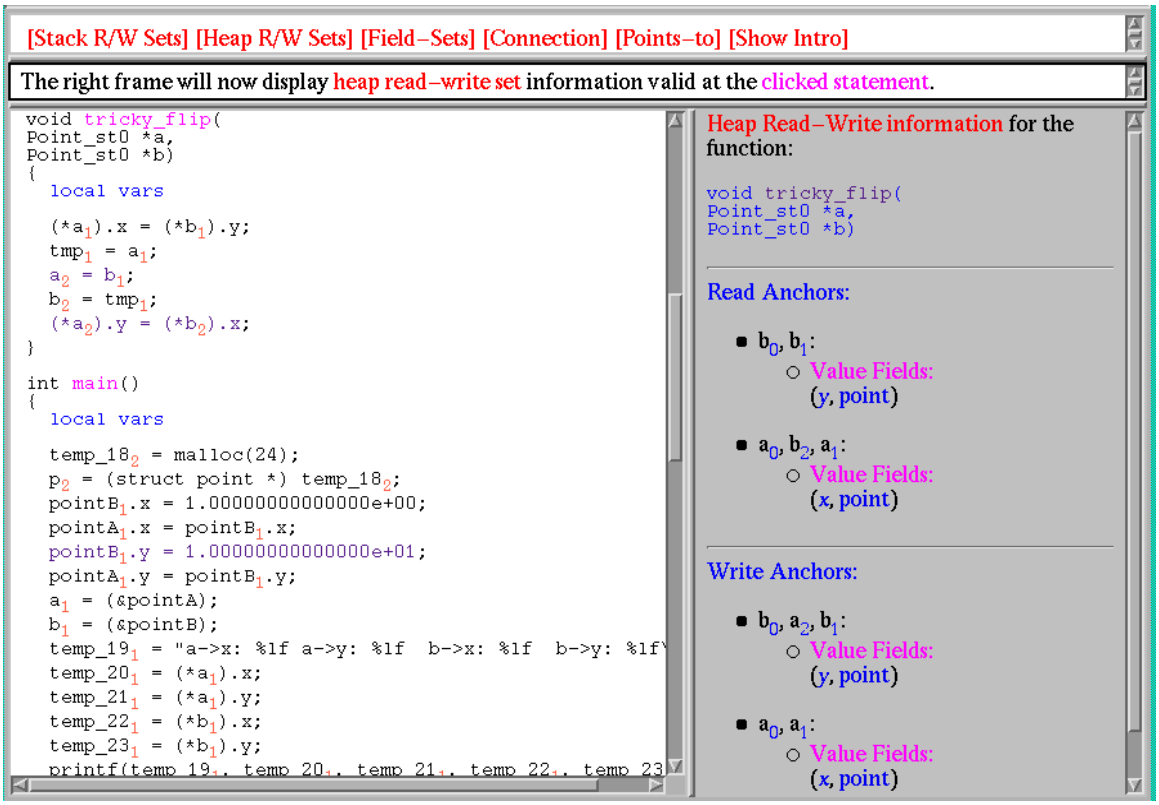


Figure 4.14: Browser Snapshot showing Heap R/W sets

The final picture shown in Figure 4.15 displays fields accessed by the function `compute_nodes` in the *em3d* benchmark. This snapshot is obtained by first clicking at *Field-Sets* in the top frame, and then at the function declaration *compute_nodes*.

Consequently, the right frame shows the set of fields accessed with respect to the anchored location `nodelist@func_entry` (internally represented as $\texttt{nodelist}_0$). Each field is again shown with the data type it belongs to and clicking on the type display its definition. As per the information, with respect to this anchor, the pointer fields `next`, `from_nodes` and `coeffs` and the scalar value fields `value` and `from_count` are *read*; only the scalar field `value` is written, and no pointer fields are written indicating that the "shape" of the data structure that came in is not changed. Similar information for any other program construct like `for`, `if` or a basic statement can be seen by simply clicking at it.

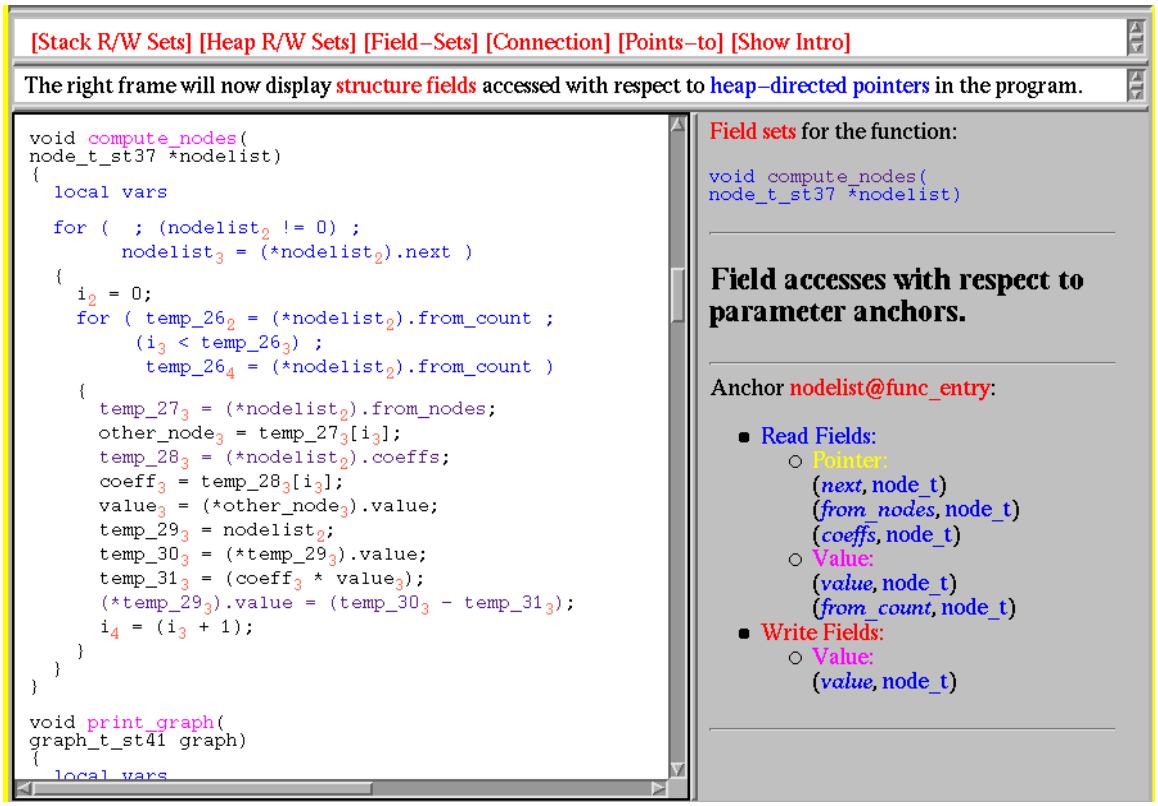


Figure 4.15: Browser Snapshot showing Field-sets

We also provide some links to facilitate program browsing itself: clicking at a function prototype, or at a function call leads one to the body of the function. A function call typically has two links: one for the CGI script that displays the flow information,

and another to the body of the function called. The former is located at the name of the function, and the latter at the position of the arguments for a simple call, or at the assignment operator for a call from an assignment statement. Also functions that are never called in the program, are shown in italics. The reader can view a demo of this tool by visiting the Web page `http://www-acaps.cs.mcgill.ca/~ghiya/BrowserDemo.html`.

4.6 Other Applications

In this section, we discuss some other compiler applications that can benefit from the read/write set information computed by our analyses. For these applications, the information collected via our read/write sets, can be communicated to other C compilers via annotations, as shown in Figure 4.12. We discuss the applications below.

4.6.1 Detecting Run-Time Constants

Dynamic compilation is currently an active area of research, with its main focus on using run-time constant information to generate more efficient code. Auslander et al. [APC+96] mention several applications where heap data structures can be run-time constants (interpreters, simulators, graphic renderers etc.). However, they rely on programmer annotation to specify which variables/data structures to consider as run-time constants. Using our heap read/write set information, in many cases run-time constant data structures can be detected automatically. Alternatively our information can be used to verify programmer annotations.

Auslander et al. use the example of a cache simulator, where the data structure pointed to by the pointer `cache` is a run-time constant. A simplified version of the example from their paper is shown in Figure 4.16. All the parameters for a particular run of the simulator (like cache size and line size) are stored in the cache data structure. Most of the fields of this data structure are not updated after being initialized by the function `initCache`. They identify the function `cacheLookup` as a candidate for dynamic compilation, based on the programmer supplied information that the pointer `cache` is a run-time constant, and that the fields `blockSize`, `numLines` and `associativity`, are not updated with respect to `cache` (only the field `tag` is variant

as denoted by the annotation 'dynamic' in the second last line of the code). Using this information, the computation of variables `tag` and `line` can be simplified, and the variable `assoc` can be detected as a run-time constant. Thus the search loop in the function, can be fully unrolled.

Using our hierarchical heap read/write information, the invariance of the fields can be automatically detected, by anchoring the pointer `cache` at program point `D`, during connection analysis. If the anchored locations `cache@D->blockSize`, `cache@D->numLines` and `cache@D->associativity`, do not belong to the heap write set of the function `main`, the three fields can be identified as run-time invariant with respect to the pointer `cache` at program point `D`.

To identify the pointer `cache` itself as a run-time constant inside the function `cacheLookup`, we can use our stack read/write sets. In this case the pointer `cache` should not belong to the stack write set of any statement in function `main`, that can follow statement `D`. However, if the pointer `cache` is not a global variable, but is passed from one function to another as a parameter, a more sophisticated analysis needs to be conducted, to identify it as a run-time constant. Such an analysis would need to track values reaching a given program point, and would also benefit from our read/write set information.

The applications that benefit from dynamic compilation (mentioned above), typically initialize a data structure once and don't update it later in the program (like the cache example). Further this data structure is mostly a stand-alone data structure in the heap, i.e., it does not share/exchange nodes with other data structures in the program. Since connection analysis disambiguates heap accesses at the data structure level, it can track reads and writes to such data structures with great accuracy, without requiring a more sophisticated and expensive heap analysis.

4.6.2 Compiler-Directed Prefetching

Recently there has been focus on prefetching accesses to recursive heap data structures [LM96]. The information needed here is which fields are potentially accessed with respect to a pointer, inside a function or a loop. This information is then used to place prefetch instructions for these heap accesses at the beginning of the given function or loop, to bring the data into the cache before it is accessed. In particular, one wants to avoid prefetching fields that will not be used. Our summary heap

```
Cache *cache;
void main()
{
     ....
     cache =
  C: initCache(size,
       lines, assoc);
  D: cacheSim();
     ....
}
```

```
cacheResult cacheLookup(void *addr)
{ /* cache is run-time constant */
  unsigned blockSize = cache->blockSize;
  unsigned numLines = cache->numLines;
  unsigned tag =
    (unsigned) addr/(blockSize * numLines);
  unsigned line =
    ((unsigned) addr/(blockSize) % numLines;
  setStructure **setArray =
    cache->lines[line]->sets;
  int assoc = cache->associativity;
  int set;
  for (set = 0; set < assoc; set++) {
    if (setArray[set]`dynamic'->tag == tag)
     return cacheHit;
}
```

Figure 4.16: Cache Example for Runtime Code Generation

read/write sets provide precisely this kind of information, as previously illustrated in section 4.5.

In Figure 4.17, we show a loop from the *bitonic sort* benchmark, used by Luk and Mowry in their study [LM96]. The loop is shown with the prefetch instructions, as presented in their paper. At the beginning of the loop, the `left` and `right` fields are prefetched with respect to the pointers `pl` and `pr`. Before the recursive calls to the `Bimerge` routine, `left` and `right` fields with respect to the `root` pointer are prefetched.

To identify the prefetch instructions to be placed at a program point, the heap read/write set information can be used. For example, if `pl@loop_entry->left` $\in$ HeapRead(L), `pl->left` should be prefetched at the beginning of the loop. Similarly, if the recursive call to `Bimerge` at program point `A`, accesses the anchored location `root@A->left`, then `root->left` should be prefetched before the call. Note that Luk and Mowry only prefetch recursive pointer fields. They assume that prefetching a recursive pointer field, should prefetch the entire node pointed to by the pointer. They mention an analysis to collect information about which pointers to prefetch, based on type and control flow information. But they do not describe their approach

in detail.

4.6.3 Identifying Memory Leaks

Another direct application of connection information is identification of potential memory leaks. When a heap-directed pointer `p` is updated, or when it goes out of scope (at function exit points), and no other *live* pointer is currently connected to it, the heap storage accessible from `p` becomes inaccessible by the program. The programmer can be warned of a potential memory leak at the given statement.

Consider for example, the code segment shown in Figure 4.18. At program point `P` after the loop, pointers `p` and `t` are connected. At program point `S`, pointer `p` is assigned a newly allocated list. This statement involves a memory leak, as the current list being pointed to by `p` has not been freed, and `p` is the last pointer to its first node. However, as per the connection information, pointer `t` still points to this list (though its last node), so a memory leak warning cannot be issued[1]. At program point `T`, pointer `t` is also updated. At this point, `t` is not connected with any pointer (its connection to `p` is killed by statement `S`). Thus the last pointer pointing to the copy of `listA` is being lost, and a memory leak warning can be issued.

4.7 Related Work on Using Pointer Analysis

As summarized in the introduction, there has been considerable work on pointer analysis. In this section we concentrate on summarizing methods that use the results of pointer analysis.

First, we consider methods that use the results of pointer analyses that are designed primarily for stack-directed pointers, and give conservative results for heap-directed pointers. Banning gave one of the seminal methods for computing side effects in the presence of aliases that arise due to call-by-reference [Ban79]. More recent algorithms based on alias analysis, for languages with explicit pointers, have been presented by Landi et. al [LRZ93] and Choi et. al [CBC93]. Using the results of

[1] However, if live variable information is available, and pointer `t` is known to be not live at program point `S`, the memory leak warning can be issued.

```
Bimerge(root,sp_r,dir)
{
  ...
  pl = root->left; pr = root->right;

  while ((pl != NIL))
  {
    prefetch(pl->left); /* Prefetch Instruction */
    prefetch(pl->right); /* Prefetch Instruction */
    prefetch(pr->left); /* Prefetch Instruction */
    prefetch(pr->right); /* Prefetch Instruction */
    if (...)
     if (...)
     { SwapValue(pl,pr); SwapRight(pl,pr);
       pl = pl->left; pr = pr->left;
     }
     else
     { pl = pl->right; pr = pr->right;
     }
     else
      if (...)
      { SwapValue(pl,pr); SwapLeft(pl,pr);
        pl = pl->right; pr = pr->right;
       }
       else
       { pl = pl->left; pr = pr->left;
       }
   }
  if ((root->left != NIL))
    {
        prefetch(root->left); /* Prefetch Instruction */
        prefetch(root->right); /* Prefetch Instruction */
        rl = root->left; rr = root->right;
        sp_l = NewNode(root->value);
     A: Bimerge(rl,sp_l,dir);
     B: Bimerge(rr,sp_r,dir);
        root->value = sp_l->value;
    }
  ...
}
```

Figure 4.17: Example Illustrating Prefetching for Recursive Data Structures

```
void singleStep(List *listA, List *listB)
{
  List *p, *t;
  if (listA == NULL)
   return;
  p = copyList(listA);
  t = p;
  while (listA->next != NULL)
  {
    listA->next->index = t->index;
    t = t->next; list = list->next;
  }
  P:
  S: p = copyList(listB); /* Memory Leak */
  T: t = p; /* Memory Leak Warning */
     ...
}
```

Figure 4.18: Identifying Memory Leaks

any points-to analysis [EGH94, WL95, Ruf95, Ste96], where all locations have names is quite straight-forward and only slight modifications of standard transformations are needed as shown in section 2.5. We assume that other compilers with points-to analyses have similar applications.

In terms of using the improved read/write sets from pointer analysis, for other analyses and transformations, the most relevant related work is of Wilson and Lam [WL95], Shapiro and Horwitz [SH97a] and Cooper and Lu [CL97]. Wilson and Lam used pointer analysis results for loop parallelization in the context of two numeric C benchmarks, *alvinn* and *specear* (Table 3.2). Shapiro and Horwitz study the effects of various flow-insensitive pointer analyses on the efficiency and precision of other compiler analyses like live variable analysis and GMOD analysis, but not on actual program transformations.

Cooper and Lu study the benefits of pointer analysis in the context of register promotion. Their work focuses on promoting address-exposed and global variables to registers inside loops when possible. They also describe a technique similar to location invariant removal to enregister pointer-based array references. Their empirical results also indicate a significant decrease in memory references for some programs, but no significant speedup. Their work can be considered a subset of our study as we do not focus on only loops, use a more precise heap analysis (they use malloc-site naming approach), and provide real runtime speedups over a state-of-the-art optimizing compiler as against comparing the number of operations executed.

Other approaches that use the results of pointer analysis have primarily focused on dependence analysis and parallelization [LH88, Gua88, HPR89, HN90, HHN94]. We will discuss these approaches in more detail in chapter 5.

Chapter 5

Detecting Parallelism in Programs with Recursive Data Structures

In the context of our overall goal of putting pointer analysis to work, so far we have shown how to compute accurate read/write sets using pointer analysis information (chapter 3), and demonstrated the benefits of the sharper read/write sets for scalar optimizations, array dependence testing and program understanding (chapter 4). In this chapter we focus on using the McCAT pointer analysis framework for a more advanced compiler application: parallelization of C programs that operate on recursive data structures.

We focus on detecting three common patterns for parallel computations that use recursive data structures: (1) function-call parallelism including parallel recursive calls on tree-like structures; (2) *forall* parallelism for loops traversing arrays of list-/tree-like structures; and (3) *foreach* parallelism for loops traversing list-/tree-like structures.

The overall goal is to detect these patterns in a sequential C program, and produce an equivalent parallel EARTH-C [HTZ+96] program. The parallel program is obtained by introducing parallel EARTH-C constructs like parallel statement sequences and *forall* loops in the original C program. The EARTH-C program can then be used as input to the EARTH-McCAT parallelizing compiler for the EARTH-MANNA parallel machine [HMT+95] (Figure 4.11, page number 103). Alternatively, parallel constructs from any other parallel dialect of C can be introduced in the original program.

In order to safely detect the above parallelism patterns in C programs, one must deal with dependencies due to scalars, dependencies due to pointers to stack-allocated objects, and dependencies due to pointers to heap-allocated objects. We use points-to based stack read/write sets to detect dependencies due to scalars and stack-directed pointers. To detect dependencies due to heap-directed pointers, we use connection-based heap read/write sets, as well as the information about the "shape" of the data structures obtained from *shape analysis* [GH96b]. For detecting the above parallelism patterns, shape information proves to be the critical component, as the patterns are based on recursive heap data structures.

The remainder of this chapter is structured as follows. In section 5.1 we introduce the three parallelism patterns in more detail. As shape information is crucial for detecting these patterns, we briefly review shape analysis in section 5.2. In section 5.3, we review the notion of dependence which is fundamental for detecting parallelism. In sections 5.4 and 5.5, we provide detailed rules and dependence tests for safely identifying the three parallelism patterns. These rules are primarily based on shape information. Some empirical results indicating how often we can successfully identify the patterns are given in section 5.6. Finally, related work is discussed in section 5.7, and a summary of the chapter is presented in section 5.8.

5.1 Parallel Patterns

In this section we illustrate the three kinds of parallelism patterns that we aim to identify. Our focus is on detecting coarse-grain parallelism in the context of function calls and loops, that perform computation on heap-based recursive data structures. The three patterns are illustrated in Figure 5.1. As per our experience, these patterns frequently arise in C programs using recursive data structures. The parallelism opportunities they offer are discussed below.

Function-call parallelism: In Figure 5.1(a), the two calls to the function `treeAdd`, respectively perform the addition for the left and right sub-trees of the tree pointed to by the pointer `t`. If the sub-trees are disjoint, the two recursive function calls access disjoint regions of the heap, and can be executed in parallel.

forall **parallelism:** Figure 5.1(b) shows an array-based loop. However, the array `list_arr` is an array of pointers, with each pointer pointing to a disjoint heap

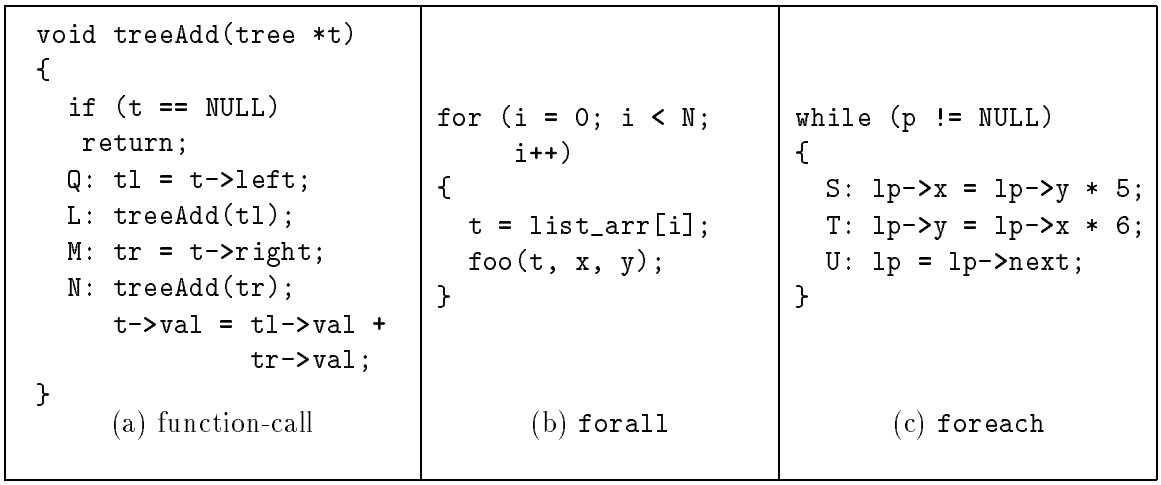


Figure 5.1: Parallelism Patterns

data structure (a list). Since each pointer points to a disjoint heap data structure, each call to function `foo` accesses a disjoint heap region, and the loop can be fully parallelized, with all iterations executing in parallel.

foreach **parallelism:** Figure 5.1(c) shows a loop traversing a linked list. The loop body consists of two parts: one that does the *computation* on the list elements, and the second that performs the *navigation* through the list. The computation part is formed by the statements `S` and `T` in the loop, while the navigation part includes the statement `U: lp = lp->next`. The pointer used to navigate through the list (`lp`), is called the *navigator*. The parallelism in this loop arises from the fact that each loop iteration visits a distinct node in the list. However, this loop cannot be considered a *forall* loop. Its iterations cannot be executed in parallel, as the loop contains an intrinsic *loop-carried* dependence due to the navigator. The navigator for the next iteration is obtained via the navigator for the current iteration. Thus the next iteration cannot start before the current iteration completes[1].

We term loops such as above, as *foreach* loops. The parallelism in a *foreach* loop can be extracted by executing it in a *staggered* fashion as shown in Figure 5.2. Here, first the navigator for the next iteration is obtained. Subsequently, the

[1]In contrast, for *forall* loops the loop index variable for any iteration can be directly computed via an arithmetic expression.

next iteration can start before the first iteration completes, and the computation phases of the two iterations can overlap. Alternatively, if the navigation overhead of the loop is minimal compared to the computation performed, the navigators could first be stored in an array of pointers of size `max_proc`, where `max_proc` is the number of processors being used, via a separate pointer-collecting loop. The original loop can then be executed as a *forall* loop. This technique is illustrated in Figure 5.3.

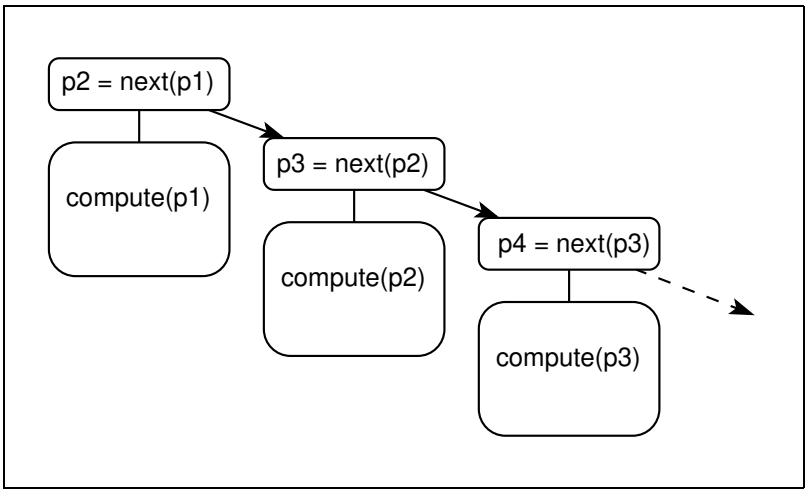


Figure 5.2: Staggered Execution of a *foreach* Loop

In order to safely detect each kind of pattern, we must analyze both the shape (treeness) of the data structure and the side-effects of the computations. Side-effects that modify the shape of the data structure, or side-effects via scalars or stack-directed pointers may also invalidate the parallelism.

First consider the impact of the shape of the recursive data structure. In Figure 5.4(a), the data structure is tree-like, and clearly any computation that reads or writes scalar fields of the tree nodes can proceed on the subtrees in parallel. For example, the recursive calls in the `treeAdd` function from Figure 5.1(a) could be computed in parallel. However, if the data structure is a DAG, as illustrated in Figure 5.4(b), then there is a dependence between the computation on the left sub-tree and the right sub-tree, as the two sub-trees have a common node. Consequently, the computations cannot be parallelized.

```
while (lp != NULL)
{
  for (i = 0; (lp != NULL) && (i < max_proc);
       lp = lp->next)
  { ptr_arr[i] = lp;
    i = i + 1;
  }
  forall (j = 0; j < i; j++)
  { lp = ptr_arr[j];
    compute(lp);
  }
}
```

Figure 5.3: Converting a *foreach* Loop to a *forall* Loop

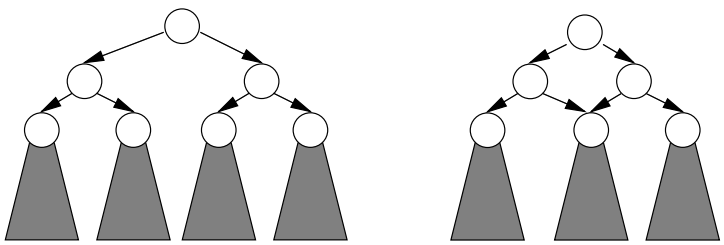


Figure 5.4: Testing for shape

To illustrate the need for other side-effect analysis, consider the example in Figure 5.1(b). If the call to `foo` writes a global variable, or a variable reachable from a parameter, then there exists a dependence between the iterations, even if the shape of the recursive data structure is tree-like. Thus, our methods for detecting the patterns need both shape and general side-effect information. The latter information is already available to us via the read/write sets. In the following sections, we first briefly describe how shape information is obtained, and then discuss in detail how it is used in conjunction with the read/write sets, for detecting parallelism.

5.2 Shape Analysis

In this section, we provide a brief description of shape analysis. Its detailed description can be found in [GH96b, Ghi95]. The goal of shape analysis is to estimate the shape of the data structure accessible from a given heap-directed pointer: is it tree-like, DAG-like or a general graph containing cycles? More specifically, the focus is on identifying unaliased tree-like and acyclic DAG-like data structures built by the program, and provide conservative estimates otherwise. Shape analysis is designed for programs that primarily use recursive data structures, or a combination of arrays and recursive data structures.

Shape analysis is composed of three *storeless* abstractions that work together and are computed together for each program point. For each heap-directed pointer, it approximates the attribute *shape*, and for each pair of heap-directed pointers it approximates the *direction* and *interference* relationships between them. The three abstractions are defined formally as follows.

Definition 5.2.1 *Given any heap-directed pointer* `p`*, the* shape *attribute* `p.shape` *is* Tree*, if in the data structure accessible from* `p` *there is a unique (possibly empty) (access) path between any two nodes (heap objects) belonging to it. It is considered to be* DAG *(directed acyclic graph), if there can be more than one path between any two nodes in this data structure, but there is no path from a node to itself (i.e. it is acyclic). If this data structure contains a node having a path to itself,* `p.shape` *is considered to be* Cycle*. Note that, as lists are a special case of tree data structures, their shape is also considered as* Tree*.*

Definition 5.2.2 *Given any two heap-directed pointers* p *and* q, direction matrix D *captures the following relationships between them:*

- D[p,q] = 1 *: An access path* possibly *exists in the heap,* from *the heap object pointed to by pointer* p, to *the heap object pointed to by pointer* q. *In this case, pointer* p *is considered to have a path to pointer* q.
- D[p,q] = 0 *: No access path exists* from *the heap object pointed to by* p, to *the heap object pointed to by* q.

Definition 5.2.3 *Given any two heap-directed pointers* p *and* q, interference matrix I *captures the following relationships between them:*

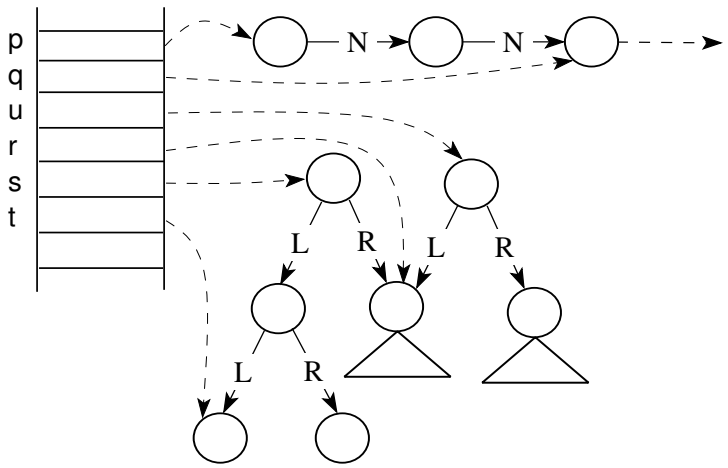
- I[p,q] = 1 *: A common heap object can be* possibly *accessed, starting from pointers* p *and* q. *In this case,* p *and* q *are considered to* interfere.
- I[p,q] = 0 *: No common heap object can be accessed, starting from pointers* p *and* q.

Direction relationships are used to actually estimate the shape attributes, while interference relationships are used for safely calculating direction relationships.

5.2.1 Illustrative Examples

We had illustrated the direction and interference abstractions in Figure 2.12 in chapter 2. For convenient reference, we illustrate them again in Figure 5.5. Part (a) shows the heap structure at a program point, while parts (b) and (c) show the direction and interference matrices for it.

In Figure 5.5, an access path exists from pointer p to q, and also from pointer s to t, so the entries D[p,q] and D[s,t] are set to one. No access path exists from pointer q to p, or from pointer r to t, so the entries D[q,p] and D[r,t] are set to zero. Further, no path exists from pointer s to u and vice versa, so both the entries D[s,u] and D[u,s] are set to zero. However starting from both u and s the heap object pointed to by r can be accessed. To indicate this, the interference matrix entries I[s,u] and I[u,s] are set to one. Finally, note that entries of the form D[p,p] are



(a) Heap Structure

	p	q	r	s	t	u
p	1	1	0	0	0	0
q	0	1	0	0	0	0
r	0	0	1	0	0	0
s	0	0	1	1	1	0
t	0	0	0	0	1	0
u	0	0	1	0	0	1

(b) Direction Matrix

	p	q	r	s	t	u
p	1	1	0	0	0	0
q	1	1	0	0	0	0
r	0	0	1	1	0	1
s	0	0	1	1	1	1
t	0	0	0	1	1	0
u	0	0	1	1	0	1

(c) Interference Matrix

Figure 5.5: Example Direction and Interference Matrices

initialized to one for all pointers. This simply indicates that the given pointer, say `p`, presently points-to a heap location.

We now demonstrate how direction relationships help estimate the shape of heap data structures. In Figure 5.6, initially the shape attribute for both `p` and `q` is Tree (`p.shape` = `q.shape` = Tree). Further `D[q,p]` is one, as there exists a path from `q` to `p` through the `next` link. The statement `p->prev = q`, sets up a path from `p` to `q` through the `prev` link. From direction matrix information it is known that a path already exists from `q` to `p`, and now a path is being set also from `p` to `q`. Thus one can deduce the creation of a cycle between heap objects pointed to by `p` and `q`. After the statement, the analysis information is as follows: `D[p,q] = 1`, `D[q,p] = 1`, `p.shape` = Cycle and `q.shape` = Cycle.

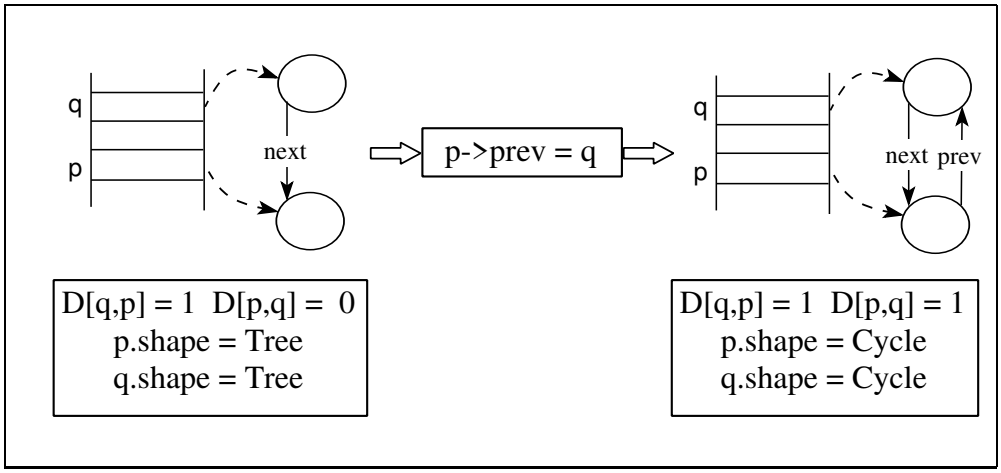


Figure 5.6: Example Demonstrating Shape Estimation

It should be noted that for a heap-directed pointer `p`, `p.shape` only abstracts the shape of the data structure *accessible* from `p` and not the overall shape of the data structure pointed to by `p`. For example, in Figure 5.7, the overall shape of the data structure pointed to by `p` and `q` is DAG. However, if only the part of the data structure accessible from `p` or `q` is considered, its shape is Tree. So the shape attribute for both `p` and `q` is Tree (`p.shape` = `q.shape` = Tree).

Knowledge about the shape of the data structure accessible from a heap-directed pointer, provides crucial information for disambiguating heap accesses originating from it. For a pointer `p`, if `p.shape` is Tree, then any two accesses of the form `p->f` and `p->g` will always lead to disjoint subpieces of the tree (assuming `f` and `g` are distinct fields). If `p.shape` is DAG, then two distinct field accesses `p->f->f` and `p->g`

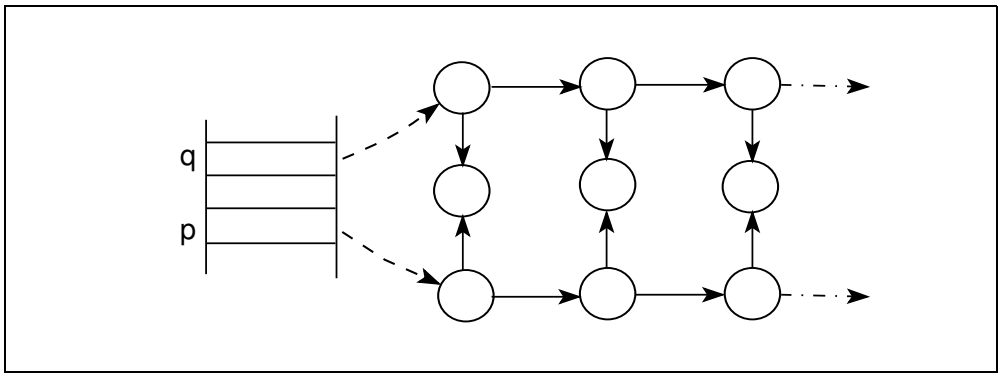


Figure 5.7: Estimating Shape with *accessibility* Criterion

can lead to a common heap object, as in Figure 5.8. However, if a dag-like structure is traversed using a sequence of links, every subsequence visits a distinct node. This information can be used to disambiguate heap accesses in different iterations of a loop, or different recursive calls, traversing such a data structure. Finally, if `p.shape` happens to be Cycle, we have effectively no information to disambiguate heap accesses originating from `p`.

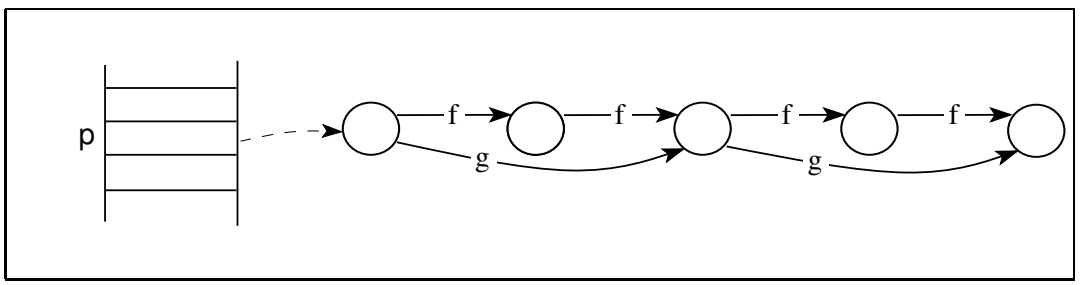


Figure 5.8: Acyclicity of Dag Data Structures

Another important point is that, shape analysis abstracts all pointers from an array of pointers, say `struct list *a[10]`, as one pointer `a`. So the relationships and attribute of this pointer, represent the merge of the relationships and attributes of all the pointers it denotes (all pointers `a[i]`). If the shape attribute of such a pointer is reported to be Tree, one is guaranteed that all array indices point to tree-like structures, completely disjoint from each other (if the structures pointed to by two array indices, say `a[i]` and `a[j]` share a node, the shape attribute for `a` is reported as DAG or Cycle).

Shape analysis has been implemented as a context-sensitive interprocedural analysis in the McCAT optimizing/parallelizing C compiler. It is a flow-sensitive analysis and collects program-point-specific information. The analysis is performed on the SIMPLE intermediate representation. The implementation is structured as a simple analysis for each basic statement, a compositional rule for each control construct, and a context-sensitive approach for handling procedure calls. Complete details can be found in [GH96b, Ghi95].

5.3 Identifying Parallelism

In this section, we briefly review the notion of *data dependence* [Wol89] between statements, which is at the heart of our algorithms for detecting the three parallelism patterns.

Definition 5.3.1 *Given two statements* `S` *and* `T` *belonging to a given function in the program, a data dependence exists from statement* `S` *to statement* `T` *if*

1. *Both* `S` *and* `T` *access a (common) memory location, and at least one of them writes to it.*
2. *There is a feasible run-time execution path from* `S` *to* `T`.

The first condition indicates that in case of a data dependence, the two statements cannot be executed in parallel without potentially violating the original semantics of the program. For example in Figure 5.9(a), there is a dependence from `S` to `T` due to the common access to the location `x`. Now if statement `T` is executed in parallel with statement `S`, it may read the value of `x` before it is written by `S`, thereby violating program semantics. Henceforth, we will use the terms data dependence and dependence interchangeably.

The second condition indicates that dependence is considered between two statements only if there is a possibility of control reaching from one statement to another.

There are three ways a data dependence can arise between the two statements:

- Statement `S` writes to a memory location that is later read by statement `T` (Figure 5.9(a)). In this case the dependence is called a *flow* dependence.

- Statement `S` reads a memory location that is later written by statement `T` (Figure 5.9(b)). In this case the dependence is called an *anti* dependence.
- Statement `S` writes to a memory location that is also later written by statement `T` (Figure 5.9(c)). In this case the dependence is called an *output* dependence.

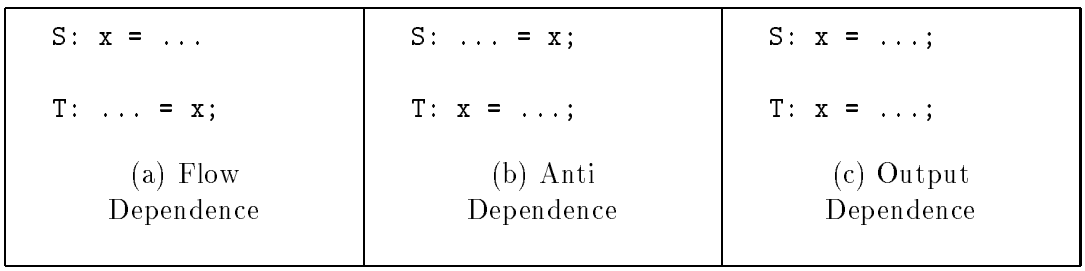


Figure 5.9: Data Dependences

It can be noted that the notion of data dependence is fundamentally tied with the sets of locations read/written by a statement. This is precisely the information computed by our read/write set analyses. In the following sections, we discuss the dependence information required for *safely* identifying our target parallelism patterns, and the tests designed to provide this information.

5.4 Function Call Parallelism

To identify if two function calls can be executed in parallel, we need to detect if there is a dependence between the statements containing them. To this end, we have developed a general dependence test that checks if a dependence exists between any two given statements in a function. The overall algorithm for the test *depTest* is outlined in Figure 5.10. The test assumes that the two statements belong to the same nest level in the SIMPLE AST, with no intervening irregular control flow constructs like `break`, `continue`, and `return`, or system calls like `exit` and `abort`. With this assumption, condition 2 of definition 5.3.1 is automatically satisfied. The focus of the dependence test is on checking for condition 1 of definition 5.3.1.

As discussed in section 5.1, a dependence can arise due to a memory access via

```
/* check if a dependence of type depType exists from stmtS to stmtT */
fun checkDependence(stmtS, stmtT, depType) =
  if (stackDependence(stmtS, stmtT, depType) == NoDep)
    return NoDep; /* no dependence */
  /* use heap analyses if dependence only due to heap */
  else if (stackDependence(stmtS, stmtT, depType) == OnlyHeap)
    if (connectionDependence(stmtS, stmtT, depType) == NoDep)
      return NoDep; /* access disjoint heap locations */
    if (shapeDependence(stmtS, stmtT, depType) == NoDep)
      return NoDep; /* access disjoint pieces of a data structure */
    return Dep; /* heap dependence cannot be disproved */
  else /* stack dependence is not only due to the symbolic location heap */
    return Dep; /* dependence cannot be disproved */
```

Figure 5.10: Checking if two Statements are Dependent

a scalar[2], a stack-directed pointer, or a heap-directed pointer. To check against all three types of memory accesses, the dependence test in Figure 5.10 proceeds in a hierarchical fashion, as explained below.

Given two statements, *stmtS* and *stmtT*, and the type of dependence to be detected (flow, anti or output), the test first tries to resolve the dependence using the points-to based stack read/write sets via the test *stackTest* (Figure 5.11). If the dependence cannot be disproved, the test then checks if the dependence is only due to heap accesses, i.e., the only conflicting memory access between the two statements is to the symbolic stack location *heap*. In this case, heap analysis information is used. First, the test *connectionTest* (Figure 5.11) is invoked. This test uses connection-based heap read/write sets to identify if the two statements access disjoint heap locations. If the test succeeds, statements are reported independent. Otherwise the test *shapeTest* is invoked to further identify if the statements access disjoint sub-pieces of a data structure.

We further illustrate the hierarchical nature of our dependence test via the code

[2]Since our focus is on programs with pointer-based data structures, we do not distinguish between different elements of an array, and consider an array reference as a scalar access.

```
/* check if the dependence can be disproved using stack read/write sets */
fun stackTest(stmtS, stmtT, depType) =
  if (depType == FlowDep)
    ISet = Write(stmtS) ∩ Read(stmtT);
  else if (depType == AntiDep)
    ISet = Read(stmtS) ∩ Write(stmtT);
  else /* if (depType == OutputDep) */
    ISet = Write(stmtS) ∩ Write(stmtT);
  if (ISet == Empty)
    return NoDep;
  else if (ISet == OnlyHeap)
    return OnlyHeap; /* only common access is to the heap */
  else
    return Dep;

/* check if the dependence can be disproved using heap read/write sets */
fun connectionTest(stmtS, stmtT, depType) =
  if (depType == FlowDep)
    ISet = HeapWrite(stmtS) ∩ HeapRead(stmtT);
  else if (depType == AntiDep)
    ISet = HeapRead(stmtS) ∩ HeapWrite(stmtT);
  else /* if (depType == OutputDep) */
    ISet = HeapWrite(stmtS) ∩ HeapWrite(stmtT);
  if (ISet == Empty)
    return NoDep;
  else
    return Dep;
```

Figure 5.11: Using Read/Write Sets to Check Dependence

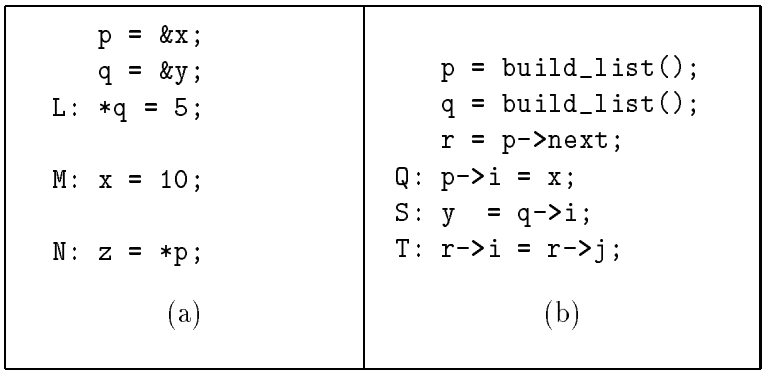


Figure 5.12: Illustrating Hierarchical Dependence Testing

segments shown in Figure 5.12. In Figure 5.12(a), first let us detect if an output dependence exists from statement `L` to statement `M`. The test *depTest* will first invoke the *stackTest*. This test will intersect the stack write sets of the two statements (Write(L) = {`y`}, Write(M) = {`x`}), get an empty *Iset*, and report no dependence. Consequently, *depTest* will terminate reporting no dependence. Second, suppose we want to detect flow dependence from statement `M` to statement `N`. In this case *stackTest* will intersect the stack write set of `M` with the stack read set of `N` (Write(M) = {`x`}, Read(N) = {`x`}), and get a non-empty *ISet* = {`x`}. Since *ISet* contains variables other than the symbolic location *heap*, *stackTest* will report a dependence. Consequently, *depTest* will terminate reporting dependence.

Now consider the code segment in Figure 5.12(b). Let us first use *depTest* to detect flow dependence from statement `Q` to statement `S`. In this case, we have the set Write(Q) = {*heap*}, and the set Read(S) = {`q`, *heap*}. Thus *stackTest* will get the set *ISet* = {*heap*}, and report dependence *only due to heap* (ONLYHEAP). Consequently, the test *connectionTest* is invoked. This test will intersect the sets HeapWrite(Q) = {`p@Q->i`} and HeapRead(S) = {`q@S->i`}, getting an empty *ISet*, as pointers `p` and `q` point to disjoint data structures (lists). Thus it will report no dependence, and *depTest* will terminate reporting no dependence.

Now let us consider the dependence relationships between statements `Q` and `T` in Figure 5.12(b). Since `p` and `r` point to the same data structure, we have the sets HeapRead(T) = {`q@Q->j, r@S->j`} and HeapWrite(T) = {`q@Q->i, r@S->i`}. Intersecting HeapRead(T) and HeapWrite(Q) gives an empty set *Iset* for the *connectionTest*, indicating no flow dependence from `Q` to `T`. However, intersecting HeapWrite(T)

and HeapWrite(Q) gives the *Iset* = {`q@Q->i`} indicating an output dependence from `Q` to `T` via the *connectionTest*. In this case, *depTest* will further need to use the *shapeTest* in order to disprove the dependence.

From the above example, we make the following important observation: *connectionTest* disproves a dependence either if the two given heap references access disjoint data structures (flow dependence from `Q` to `S`), or disjoint fields within heap objects (nodes) belonging to the same data structure (flow dependence from `Q` to `T`). Thus *shapeTest* is invoked only when the two heap references access the same field within nodes belonging to the same data structure (output dependence from `Q` to `T`). Hence, the objective of *shapeTest* is to identify if the two heap references access disjoint nodes in a given data structure.

5.4.1 Shape Dependence Test

Unlike points-to and connection information, the shape analysis information is not explicitly encoded as read/write sets, and the dependences cannot be detected by a simple intersection of these sets. The shape dependence test relies on the shape of the data structure being traversed, and the *access paths* used to reach particular node(s) in the data structure. The access paths are computed with respect to a given node in the data structure, pointed to by the *anchor* pointer. An *anchor* pointer is a pointer that points to a fixed node in the data structure over the program region starting from the statement that defines it, and ending at the statement `T`, where a dependence is being checked from some statement `S` to statement `T`. Once the access paths are computed with respect to the *anchor*, dependence is resolved by checking if starting from the *anchored* node, the two paths can lead to the same node, in view of the shape of the data structure.

The shape dependence test is outlined in Figure 5.13. It first collects the sets of pointers *pSetS* and *pSetT* that access the heap, respectively for *stmtS* and *stmtT*. These sets are computed using the read/write set information, as shown in Figure 5.14. The shape dependence test is performed on each pair of pointers (*ptrS*, *ptrT*) from the two sets, and no dependence is reported only if the test succeeds on each pair. For a given pair, if the shape attribute of either pointer is found to be cyclic, a dependence is reported and the test terminates. If the shape attributes are acyclic, anchor-based access paths are constructed, as explained below.

```
fun shapeTest(stmtS, stmtT, depType) =
 [pSetS, pSetT] = heapAccessPtrs(stmtS, stmtT, depType);
 foreach pair (ptrS, ptrT) ∈ pSetS×pSetT
   if (isCycle(ptrS.shape) or isCycle(ptrT.shape))
     return DEP; /* Cyclic data structures */
   [defS, defT] = getDefChains(ptrS, stmtS, ptrT, stmtT);
   anchorStmt = findAnchor(defS, defT);
   if (anchorStmt != NULL)
     anchor = getRef(anchorStmt, lhs);
   if (!anchor or isCycle(anchor.shape))
     return DEP; /* cannot find anchor or its shape is cyclic */
   [pathS, pathT] = getPathExprs(anchor, defS, defT);
   if (fieldsUpdatedBetween(pathS, stmtS, pathT, stmtT, anchorStmt))
     return DEP; /* structural modification involved */
   if (checkPathExprs(pathS, stmtS, pathT, stmtT, anchor) == DEP)
     return DEP; /* path exprs indicate a possible conflict */
 return NODEP; /* no dependence detected */
```

Figure 5.13: Using Shape Analysis for Dependence Detection

```
/* get the set of pointers that access the heap for the two stmts */
fun heapAccessPtrs(stmtS, stmtT, depType) =
 if (depType == FlowDep)
   pSetS = getHeapAccessPtrs(stmtS, Write);
   pSetT = getHeapAccessPtrs(stmtT, Read);
 else if (depType == AntiDep)
   pSetS = getHeapAccessPtrs(stmtS, Read);
   pSetT = getHeapAccessPtrs(stmtT, Write);
 else /* if (depType == OutputDep) */
   pSetS = getHeapAccessPtrs(stmtS, Write);
   pSetT = getHeapAccessPtrs(stmtT, Write);
 return [pSetS,pSetT];

/* get the set of pointers that read/write to the heap at stmt */
fun getHeapAccessPtrs(stmt, accessType) =
 pSet = Empty;
 if (accessType == Write)
   foreach ptr ∈ Read(stmt) /* pointers used in the given stmt */
     foreach anchor ∈ HeapWrite(stmt) /* heap anchors written by stmt */
       if (ptr, anchor) ∈ Connect(stmt) /* ptr is connected with anchor */
         pSet = pSet ∪ {ptr}; /* include it in the set */
 if (accessType == Read)
   foreach ptr ∈ Read(stmt)
     foreach anchor ∈ HeapRead(stmt) /* heap anchors read by stmt */
       if (ptr, anchor) ∈ Connect(stmt)
        pSet = pSet ∪ {ptr};
 return pSet;
```

Figure 5.14: Collecting Pointers that Access the Heap

Constructing Access Paths using Definition Chains

The first step in the construction of access paths is computation of definition chains *defS* and *defT* for the pointers *ptrS* and *ptrT*. Definition chains are constructed by recursively traversing the reaching definitions of the given pointers. The algorithm is outlined in Figure 5.15. We illustrate it in the following example.

```
/* get definition chains for uses of the two pointers at the given stmts */
fun getDefChains(ptrS, stmtS, ptrT, stmtT) =
  defS = getDefChain(ptrS, stmtS, initChain);
  defT = getDefChain(ptrT, stmtT, initChain);
  return [defS, defT];

/* get definition chain for use of the pointer ptr at the statement stmt */
fun getDefChain(ptr, stmt, chain) =
  defList = reachingDefs(ptr, stmt);
  if (listNext(defList) != NULL || !definiteDef(defList))
    return chain; /* more than one definition or a may-definition */
  defStmt = getStmt(defList);
  /* stmts not in the same nest level or intervened by irreg. control flow */
  if (!sameNestLevel(defStmt, stmt) || (irregControlBetween(defStmt, stmt)))
    return chain; /* def at defStmt conditionally reaches its use at stmt */
  newChain = makeChain(defStmt);
  appendToChain(chain, newChain); /* include the def in the def-chain */
  if (isFunctionHeader(defStmt))
    return chain; /* traversal has reached the function header */
  rhsRef = getRef(defStmt, rhs);
  useVar = getBaseVar(rhsRef); /* base variable used by the defStmt on rhs */
  chain = getDefChain(useVar, defStmt, chain); /* recursive traversal */
  return chain; /* return the definition chain */
```

Figure 5.15: Computing Definition Chains

Consider the construction of the definition chain for the use of pointer `tr` at the function call statement `N: treeAdd(tr)` in Figure 5.1(a) (re-illustrated in Figure 5.16). The definition that reaches this use is from the statement `M: tr = t->right`. So this statement is put in the definition chain. Next, the traversal looks for definitions that reach the use of pointer `t` at the statement `M`. In this case the definition

comes from the function header, which is appended to the definition chain. Since the traversal cannot proceed any further up, it stops. Similarly, the definition chain for the use of pointer `tl` at statement `L` would consist of the statement `Q` and the function header.

```
void treeAdd(tree *t)
{
  if (t == NULL)
   return;
  Q: tl = t->left;
  L: treeAdd(tl);
  M: tr = t->right;
  N: treeAdd(tr);
     t->val = tl->val + tr->val;
}
```

Figure 5.16: Function-call Parallelism

In general, the definition chain traversal stops when either of the following conditions apply: (i) it has reached the function header, (ii) more than one reaching definition is found, (iii) the definition found *conditionally* reaches the use, i.e., the definition statement is either nested in a conditional or there exists an irregular control flow construct like `break`, `return` etc., between the definition statement and the statement that uses it; or (iv) there is only one reaching definition, but it is not a *definite* definition of the relevant pointer.[3] The last three conditions are required because we want to construct only one *definite* access path for a pointer and not a set of *possible* paths. This is done for efficiency reasons, as comparing a set of paths can be expensive and further is less likely to disprove a dependence.

Finding a Common Anchor

Once the definition chains are constructed, the next step is to find an *anchor* pointer, with respect to which both the pointers under consideration can be defined. In our example above, the pointer `t` can be considered as an anchor pointer, as both pointers `tl` and `tr` can be defined in terms of `t`. This is infered from the fact that both the pointers have a common definition of the pointer `t` in their definition chains.

[3] A definition is considered as *definite*, if it either involves a scalar variable or a pointer indirection with the pointer *definitely* pointing to a single stack location.

Now, using the function header definition of `t` as the anchor, the definition chains of pointers `tl` and `tr` are traversed to construct their respective access paths with respect to the anchor `t`, giving the paths `t->left` and `t->right`. The algorithms for finding the anchor and constructing access paths are outlined in Figure 5.17.

Above we have explained how access paths with respect to an anchor are constructed. This is the crucial information required for the shape dependence test. The access paths are given to the function *checkPathExpr*, which detects if they definitely lead to disjoint parts of the data structure. Before reaching this step, the dependence test may terminate prematurely, reporting dependence, in the following cases: (i) the definition chains of the two pointers under consideration, do not contain a common anchor, (ii) the fields used in the paths are updated between the anchor definition point and the statements under consideration, or (iii) the shape attribute of the anchor is cyclic. In the first case, the access paths cannot be constructed. In the second case, the shape of the data structure being traversed can change in between, and access paths cannot be *safely* used for disambiguation. This check is performed using connection-based heap read/write sets as shown in Figure 5.18. For the third case, the *anchor* pointer points to a cyclic data structure, so the access paths cannot be effectively used to disprove the dependence.

Comparing Access Paths for Dependence Detection

We now explain the dependence checks done by the function *checkPathExpr* shown in Figure 5.19, which is the core component of the shape dependence test. As discussed above, the *shapeTest* reaches this function only if the shape of the data structure being traversed is Tree or DAG, and the traversal fields are not being modified. The tests performed on the access paths differ according to the types of the statements involved. A statement can either be a call statement (of type CallStmt), indicating that it involves a function call that accesses the heap; or it can be a simple statement (of type SimpleStmt) indicating that it is a simple assignment statement involving indirect references to the heap but no function calls.

The above distinction is made because of the following reason. Suppose an access path for a statement leads to some node in the data structure, say *nodeA*. So the statement uses a pointer pointing to *nodeA*, say `ptrA`. If the statement is a simple statement, it can use the pointer `ptrA` to only access the fields of the specific node *nodeA*. This limitation arises from the fact that our SIMPLE intermediate representation allows only one level of indirection at a statement. Now if the statement is a call

```
/* find the anchor for the two def chains: an anchor exists if the two chains
 * contain a common defStmt, and is the variable defined by this defStmt */
fun findAnchor(defS, defT) =
  while (defS != NULL)
    defStmtS = getStmt(defS);
    while (defT != NULL)
      defStmtT = getStmt(defT);
      if (defStmtS == defStmtT)
        anchorStmt = defStmtS;
        return anchorStmt; /* anchor stmt found */
      defT = listNext(defT);
    defS = listNext(defS);
 return NULL; /* cannot find a common anchor */

/* construct access paths with respect to the anchor */
fun getPathExprs(anchor, defS, defT) =
  pathS = getPathExpr(anchor, defS);
  pathT = getPathExpr(anchor, defT);
  return [pathS,pathT];

/* construct access path with respect to the anchor */
fun getPathExpr(anchor, defChain) =
  path = NULL;
  while (defChain != NULL)
    defStmt = getStmt(defChain);
    rhsRef = getRef(defStmt, rhs); /* get rhs: p->next for stmt t=p->next */
    /* append p->next to the path: path t->left becomes p->next->left */
    appendToPath(path, rhsRef);
    lhsRef = getRef(defStmt, lhs);
    if (anchor == lhsRef)
      return path; /* reached the definition point of anchor itself */
    defChain = listNext(defChain);
  return NULL; /* should not reach this point, if the anchor is valid */
```

Figure 5.17: Using Definition Chains to Find the Anchor and the Access Paths

```
/* if fields in pathS/pathT are updated between anchorStmt and stmtS/stmtT */
fun fieldsUpdatedBetween(pathS, stmtS, pathT, stmtT, anchorStmt) =
 anchor = getRef(anchorStmt, lhs);
 foreach stmt in [anchorStmt:stmtS] /* stmts between anchorStmt & stmtS */
  if (fieldsUpdated(pathS, stmt, anchor))
   return TRUE;
 foreach stmt in [anchorStmt:stmtT] /* stmts between anchorStmt & stmtT */
  if (fieldsUpdated(pathT, stmt, anchor))
   return TRUE;
 return FALSE;

fun fieldsUpdated(path, stmt, anchor) =
 /* get connection analysis anchor for the anchor pointer at the
  * given statemet stmt, to access heap read/write sets */
 connectionAnchor = getConnectionAnchor(stmt, anchor);
 foreach field in path
  if (connectionAnchor->field ∈ HeapWrite(stmt))
   return TRUE;
 return FALSE;
```

Figure 5.18: Checks for Field Updates

```
fun checkPathExprs(pathS, stmtS, pathT, stmtT, anchor) =
  case type(stmtS) of
    < CallStmt > =>
      case type(stmtT) of
        < CallStmt > =>
          if (isDag(anchor.shape))
            return DEP; /* DAG shape is not useful with two call stmts */
          else /* shape is Tree */
            return(disjointPaths(pathS, pathT) ? NODEP : DEP);
        < SimpleStmt > =>
          if(subPath(pathT, pathS))
            return NODEP; /* sufficient condition */
          if (isTree(anchor.shape))
            return(disjointPaths(pathS, pathT) ? NODEP : DEP);
    < SimpleStmt > =>
      case type(stmtT) of
        < CallStmt > =>
          if(subPath(pathS, pathT))
            return NODEP;
          if (isTree(anchor.shape))
            return(disjointPaths(pathS, pathT) ? NODEP : DEP);
        < SimpleStmt > =>
          if (isTree(anchor.shape))
            return(!equivPaths(pathT, pathS) ? NODEP : DEP);
          else /* anchor shape is DAG : one is subpath of another */
            if (subPath(pathT, pathS) or subPath(pathS, pathT));
              return NODEP;
            return DEP;
```

Figure 5.19: Comparing Access Paths for Dependence Detection

statement, it can pass `ptrA` as a parameter and the callee function can access fields of all nodes accessible from `ptrA`. For example, in Figure 5.16, the call statement `N: treeAdd(tl)` accesses all nodes in the left sub-tree via `tl`, while the simple statement `t->val = tl->val + tr->val`, only accesses a field of the specific node pointed to by `tl`. This distinction is important in using access paths and shape information for dependence detection.

Operations to Compare Access Paths

The input to the function *checkPathExprs* consists of two statements, *stmtS* and *stmtT*, and the respective access paths, *pathS* and *pathT*, expressed with respect to the anchor pointer. The function uses three operations to compare the access paths:

equivPaths(pathS, pathT) : This function checks if the two access paths are equivalent, i.e., consist of the same sequence of field accesses. For example the access path `t->left->right` is equivalent to the access path `t->left->right`, but *not equivalent* to the access paths `t->left` or `t->left->left`. Two equivalent access paths always lead to the same node. Equivalent paths are illustrated in Figure 5.20(a).

subPath(pathS, pathT) : This function checks if the access path *pathS* is a proper sub-path of the access path *pathT*, i.e., *pathS* contains k field accesses less than *pathT*, and is equivalent to the access path obtained by removing the last k field accesses from *pathT*, where $k \geq 1$. For example, the access path `t->left` is a proper sub-path of the access path `t->left->right`, but not of the access paths `t->left` or `t->right`. For acyclic data structures (anchor.shape is Tree or DAG), if an access path is a proper sub-path of another path, the two paths lead to disjoint nodes. Further, the node reached via the former path *cannot* be accessed from the node reached via the latter path. Proper sub-paths are illustrated in Figure 5.20(b). Here path p1 is a proper sub-path of path p2: p1 is equivalent to p2', which is a proper sub-path of p2. Note that the node reached via p1 cannot be accessed from the node reached via p2.

disjointPaths(pathS, pathT) : This function checks three conditions: (i) the access path *pathS* is not equivalent to the access path *pathT*, (ii) *pathS* is not a proper sub-path of *pathT*, and (iii) *pathT* is also not a proper sub-path of path *pathS*. For example, `t->left` and `t->right` are disjoint paths, while `t->left` and `t->left->right` are not. For tree-like data structures (anchor.shape is Tree), disjoint paths not only lead to disjoint nodes, but also one node cannot be accessed from the other. Thus disjoint paths lead to disjoint sub-pieces of tree-like data structures. This situation is illustrated in Figure 5.20(c).

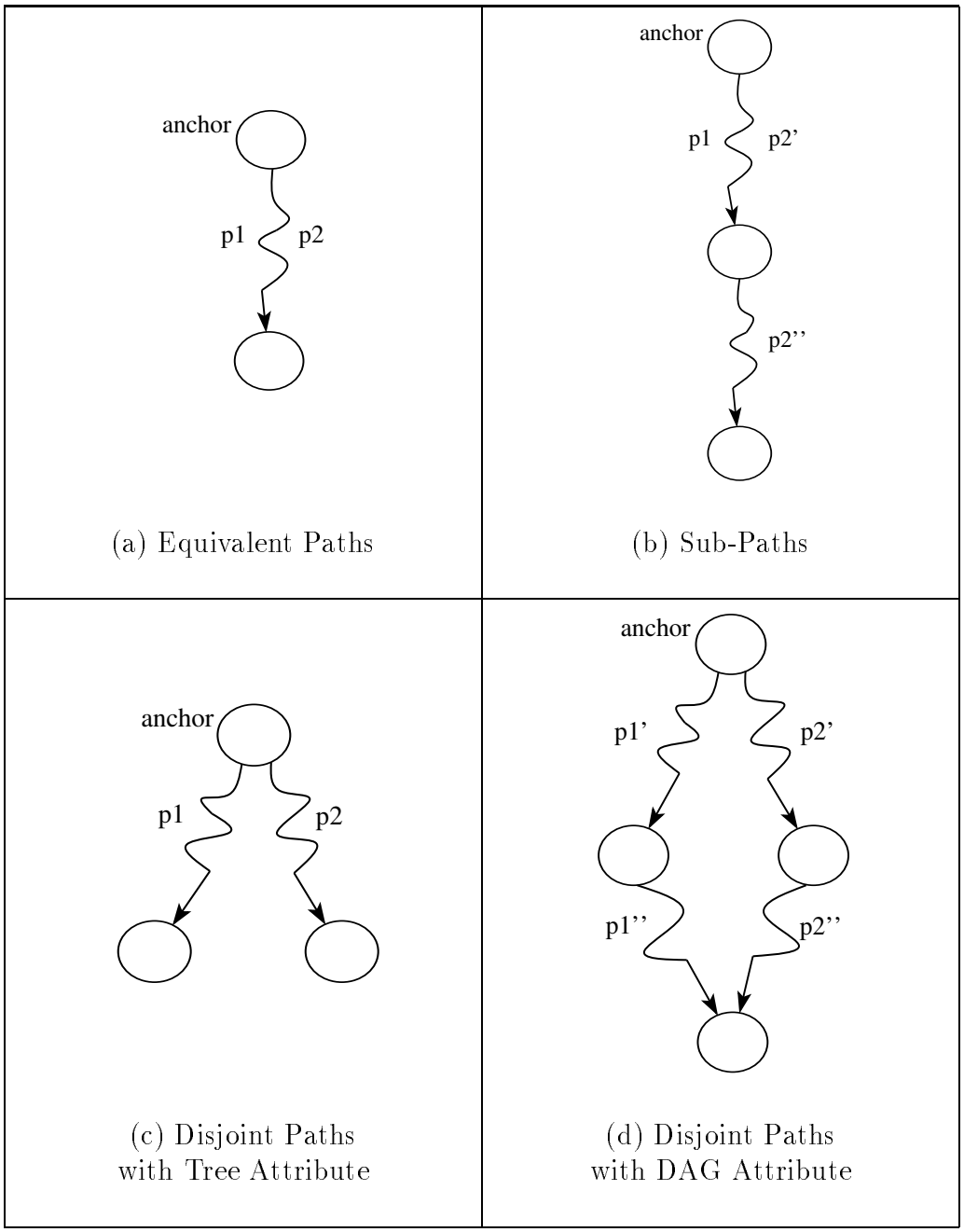


Figure 5.20: Illustrating Access Path Comparisons

If the data structure is dag-like (anchor.shape is DAG), disjoint paths can lead to a common node. For example, if `left` and `right` links of the node pointed to by the anchor pointer `t`, point to the same node (giving a DAG), the disjoint paths `t->left` and `t->right` will lead to the same node. This situation is further illustrated in Figure 5.20(d). Here the shape attribute of the anchor is DAG. The paths p1 and p2 are disjoint paths. Their sub-paths p1' and p2' lead to disjoint nodes. However, the remaining sub-paths p1" and p2", lead back to a common node. On the contrary, with Tree shape attribute, starting from two distinct nodes, no two paths can lead to a common node.

Different Cases for Dependence Detection

We now discuss the different cases and the associated disambiguation rules given in the function *checkPathExprs* in Figure 5.19 (page number 142). We have four cases depending on the types of the two statements, *stmtS* and *stmtT*, as explained below.

Case 1: Both statements are call statements

As explained before, call statements can access whole sub-pieces of a data structure as opposed to a single node. Thus for two call statements to be independent, the access paths should lead to disjoint sub-pieces of the data structure. This requires that the two access paths are disjoint and the shape attribute of anchor is Tree, as shown in the function *checkPathExpr* (Figure 5.19). This is the case for our example based on the `treeAdd` function shown in Figure 5.16. The two access paths are `t->left` and `t->right`, which are disjoint, and the shape attribute of the anchor pointer `t` is Tree. Thus the two calls to `treeAdd` are independent and can be executed in parallel.

Case 2: stmtS is a call statement and stmtT is a simple statement

Here *stmtS* can access a sub-piece of the data structure, while *stmtT* can access only fields in a specific node of the data structure. Thus for the two statements to be independent, *pathS* should lead to a node, which is disjoint from the node corresponding to *pathT*, and also cannot reach the latter node. This can be guaranteed if *pathT* is a proper sub-path of *pathS*. The shape attribute of the anchor can be either Tree or DAG in this case. For example, if *pathS* is `t->left` and *pathT* is simply `t`, the proper sub-path condition is satisfied, and *stmtS* cannot access the node accessed by *stmtT*. If shape attribute of the anchor is Tree, the statements will be independent also for the case when *pathS* and *pathT* are disjoint.

Case 3: stmtT is a call statement and stmtS is a simple statement

This case is analogous to the case 2.

Case 4: Both statements are simple statements

In this case both statements can respectively access only some specific node of the data structure. We simply need to check that *pathS* does not lead to the same node as *pathT*. This condition is satisfied, if *pathS* is not equivalent to *pathT*, when shape attribute for the anchor is Tree. With DAG attribute, we need to check for the stronger condition that one of the access paths is a proper sub-path of another.

For example, let *pathS* be `t->right->left` and *pathT* be `t->left->right`. The access paths are not equivalent. With shape attribute as Tree, the two access paths cannot lead to the same node. With DAG attribute they can lead to the same node. However, the access paths `t->left` and `t->left->right` can be proven to lead to disjoint nodes even with DAG attribute. This is because the common sub-path in the paths leads them to the same node, and then the additional field access in the latter path, leads it to a distinct node as the data structure is acyclic.

Above, we have described our overall strategy for detecting dependence between *two given statements* in a function, and discussed in detail how to use shape information for dependence testing. The dependence test can be used for any two statements in the program, but is particularly designed for statements containing recursive function calls accessing heap data structures. To indicate statements that can be executed in parallel, we use our dependence test (*depTest*) during the DDG (data dependence graph) construction phase of the EARTH-McCAT compiler [TGHG97]. The DDG is then used to identify statements/function calls that can be executed in parallel, and to partition the program into threads. Sharper dependence information from our tests, leads to a more accurate DDG, which exposes more parallelism and enables better thread partitioning.

5.5 Loop Parallelism

In this section, we present techniques to identify loop-level parallelism, in the form of *forall* and *foreach* loops traversing recursive heap data structures. For function-call parallelism, we simply need to check if a dependence exists between two given statements in the program. For finding loop parallelism, we need to additionally detect the presence of *loop-carried* dependences (henceforth refered to as LCDs). Two statements in a loop have an LCD, if a memory location accessed by one statement in a given iteration, is accessed by the other statement in a future iteration, with one of the accesses being a write access.

The following loop illustrates an LCD. In this loop, the reference `p->next->i` in the current iteration writes to the memory location, read by the reference `p->i` in the next iteration, thereby inducing an LCD.

```
while (p->next != NULL)
{
  p->next->i = p->i;
  p = p->next;
}
```

The presence of LCDs in a loop indicates that its iterations are not independent, and hence cannot be executed in parallel. Our particular focus is on finding heap-based *forall* and *foreach* loops (described in section 5.1). A *forall* loop should not have any LCD, while a *foreach* loop can only involve an LCD with respect to the navigator. For example, in the *foreach* loop shown in Figure 5.1(c) (re-illustrated in Figure 5.21), the only LCD present is induced by the statement that advances the navigator (`U: lp = lp->next`). The statement `U` writes to the navigator `lp` in the current iteration, which is read by `U` in the next iteration.

```
while (lp != NULL)
{
  S: lp->x = lp->y * 5;
  T: lp->y = lp->x * 6;
  U: lp = lp->next;
}
```

Figure 5.21: Example *foreach* Loop

Considering these constraints, we first describe the method for detecting *foreach* loops, and then explain how it can be adapted to detect *forall* loops.

Given a loop, we identify if it is a heap-based *foreach* loop, using the following steps:

1. **Good Loop Detection:** This step identifies if the given loop traverses a recursive heap data structure in a *regular* fashion.

2. **Detecting Heap-based LCDs:** This step checks for the presence of LCDs with respect to heap accesses. If no heap-based LCDs are detected, the loop is labeled as a *heap foreach* loop.

3. **Detecting Stack- and Array-based LCDs:** This step uses existing techniques implemented in the McCAT C compiler to detect LCDs induced by stack or array accesses in the loop. If the only LCDs detected are due to accesses to the navigator, the loop is labeled as a *real foreach* loop. In this case, the loop can be transformed as shown in Figure 5.2 or Figure 5.3.

We now explain the steps in detail.

5.5.1 Good Loop Detection

This is a pre-processing step, which detects potential heap-based *foreach* loops in the program. It is required so that we do not incur the overhead of detecting LCDs for each loop in the program. The criteria used to label a loop as a good loops are as follows:

1. The loop body involves read/write accesses to heap locations.

2. The loop body is free from irregular control flow constructs such as `break`, and `return` statements, or system calls such as `exit` and `abort`. Thus control can exit the loop only from the *loop condition test.*

3. A navigator can be identified for the loop.

The first condition ensures that the loop is heap-based. We verify this condition by using our points-to based stack read/write sets for the loop. If a loop `L` involves read/write accesses to the heap, the symbolic stack location *heap* must belong to the stack read or write set of the loop, i.e., *heap* $\in$ Read(L) $\cup$ Write(L).

The second condition is required to ensure the correctness of the parallelizing transformations illustrated in Figure 5.2 and Figure 5.3. In either transformation, an iteration of the loop can start executing before the completion of previous iterations. If the loop can terminate prematurely in an iteration via one of the constructs shown

above, the side effects of future iterations overlapping with it, can alter the intended execution semantics of the program. Besides ensuring the correctness of parallelizing transformations, this condition also proves to be useful for detecting parallelism itself. We will discuss this further in section 5.5.2.

Finally, the third condition is required to detect that the loop actually *navigates* a recursive heap data structure in a regular fashion. Further, identification of the navigator is critical for detecting heap-based LCDs. To detect if two heap references in the loop can induce an LCD, we first need to express them in terms of the navigator (section 5.5.3). We now describe our method for identifying the navigator for a loop.

Identifying the Navigator

The overall algorithm for identifying the navigator is outlined in the functions *identifyNavigator* and *findNavigator* shown in Figure 5.22. The process is closely related with the variables used in the *loop test*: the test performed to determine if the next iteration of the loop should be executed. The function *identifyNavigator* first tries to find the navigator starting from the left hand side variable in the test. In case of failure, it tries the right hand side variable. The order is completely arbitrary. Note that in our SIMPLE representation, a condition test only involves scalars or constants. If the reference on lhs/rhs of the loop test involves a constant (`lp != NULL`) or a loop-invariant variable (`i < N`), then the corresponding reference is not considered for navigator identification, as shown in the rule in Figure 5.22.

For a given variable in the loop test, say `testVar`, the function *findNavigator* proceeds as follows. First, a definition chain is constructed for the use of `testVar` in the loop test. The function *getLoopDefChain* defined in Figure 5.23, is used for this purpose. This function is similar to the function *getDefChain* defined in Figure 5.13, and constructs the definition chain by recursively traversing reaching definitions. However, during this traversal, it only considers the definitions that arise from a statement within the given loop (loop-resident definitions). It terminates when one of the following conditions apply:

1. No loop-resident definition is found.
2. More than one loop-resident definition is found.
3. The loop-resident definition found, is not a *definite* definition.

```
/* find the navigator for the given loop if one exists */
fun identifyNavigator(loopStmt) =
 cond = loopStmt.cond; /* loop condition test */
 navigator = findNavigator(cond.lhsvar, loopStmt);
 if (navigator != NULL) /* lhs var succeeds */
  return navigator;
 else /* try rhs var in the loop test */
  return (findNavigator(cond.rhsvar, loopStmt));

/* find the navigator for the given loop with respect to the var */
fun findNavigator(var, loopStmt) =
 defN = getLoopDefChain(var, loopStmt, loopStmt, initChain);
 if (defN == NULL || recFlag(defN) != RECUR)
   return NULL; /* no recurrence exists in the definition chain */
 pathT = getPathExpr(loopStmt.cond.var, defN); /* test expression */
 pathN = getNavExpr(pathT); /* navigator expression */
 varN = getBaseVar(pathN); /* navigator */
 if (fieldsUpdated(pathN, loopStmt, navigator))
   return NULL; /* structural modification involved */
 loopStmt.navigatorExpr = pathN;
 loopStmt.navigator = varN;
 return (varN);
```

Figure 5.22: Rules for Identifying the Navigator

4. The loop-resident definition found, *conditionally* reaches its use, i.e., is embedded inside a conditional statement.

5. A loop-resident definition is encountered for the second time.

In the last case a recurrence is reported (by setting the recurrence flag). If the traversal terminates without finding a recurrence, it returns NULL, indicating that a navigator cannot be detected with respect to the given `testVar`. Note that conditions 2, 3 and 4 ensure that *all the statements included in the definition chain are definitely executed in each loop iteration.*

If the function *getloopDefChain* reports a recurrence, it indicates the presence of a variable in the loop whose value for the next iteration is defined in terms of its current value. Such a variable is a potential candidate for being a navigator. In this case, the definition chain is used to construct an access path for the loop test variable `testVar`. This access path is called the *test expression* and the base variable in the path is called the *navigator*. The test expression indicates how the `testVar` for the next loop test is obtained from the navigator for the current iteration. The component of the access path, contributed by the definition(s) involved in the recurrence, is called the *navigator expression*. It indicates how the navigator for the next iteration is obtained in terms of the current navigator. We illustrate these concepts via examples below.

First, consider the loop in Figure 5.24(a). The loop test variable in this case is the pointer `lp`. Its loop-resident definition comes from the statement `S: lp = lp->next`. So it is added to the definition chain. Next, the loop-resident definition for the use of `lp` at `S` is sought. It again happens to be the statement `S`. Here the definition chain construction terminates and a recurrence is reported. The definition chain gives the access path `lp->next`, which is the test expression for the loop, and the base pointer for the expression, `lp`, is the navigator. Here, since the navigator is identical to the loop test variable, the navigator expression is same as the test expression.

The loop shown in Figure 5.24(b), shows an instance where the navigator is not same as the loop test variable. Here, the definition chain consists of statements `V` and `U`, with the use of `lp` at statement `U` producing the recurrence. The test expression for the loop is `lp->next->i`, the navigator is the pointer `lp` (not the loop test variable `index`), and the navigator expression is `lp->next`.

```
/* get definition chains only considering the loop-resident defs */
fun getLoopDefChains(ptrS, stmtS, ptrT, stmtT, loopStmt) =
  defS = getLoopDefChain(ptrS, stmtS, loopStmt, initChain);
  defT = getLoopDefChain(ptrT, stmtT, loopStmt, initChain);
  return [defS,defT];

/* get definition chain for use of the ptr at stmt inside the loop loopStmt */
fun getLoopDefChain(ptr, stmt, loopStmt, chain) =
  defList = getLoopDefs(ptr, stmt, loopStmt); /* get all loop-resident defs */
  if (defList == NULL)
    return chain; /* no loop-resident defs found */
  if (listNext(defList) != NULL || !definiteDef(defList))
    return chain; /* more than one definition or a may-definition */
  defStmt = getStmt(defList);
  if (reachesConditionally(defStmt, stmt))
    return chain; /* def at defStmt conditionally reaches its use at stmt */
  if (checkRecurrence(defStmt, chain) == RECUR)
    return chain; /* recurrence found: so stop traversal */
  newChain = makeChain(defStmt);
  appendToList(chain, newChain); /* include the def in the def-chain */
  rhsRef = getRef(defStmt, rhs);
  useVar = getBaseVar(rhsRef); /* base variable used by the defStmt on rhs */
  /* recursive traversal */
  chain = getLoopDefChain(useVar, defStmt, loopStmt, chain);
  return chain; /* return the definition chain */

/* check if the definition at newStmt already exists in the defChain */
fun checkRecurrence(defStmt, defChain) =
  tChain = defChain
  while (tChain != NULL)
    tStmt = getStmt(tChain);
    if (tStmt == defStmt)
      recFlag(defChain) == RECUR;
      return RECUR; /* recurrence found */
    tChain = listNext(tChain);
  return NORECUR; /* no recurrence */
```

Figure 5.23: Computing Definition Chains for Loop-resident Definitions

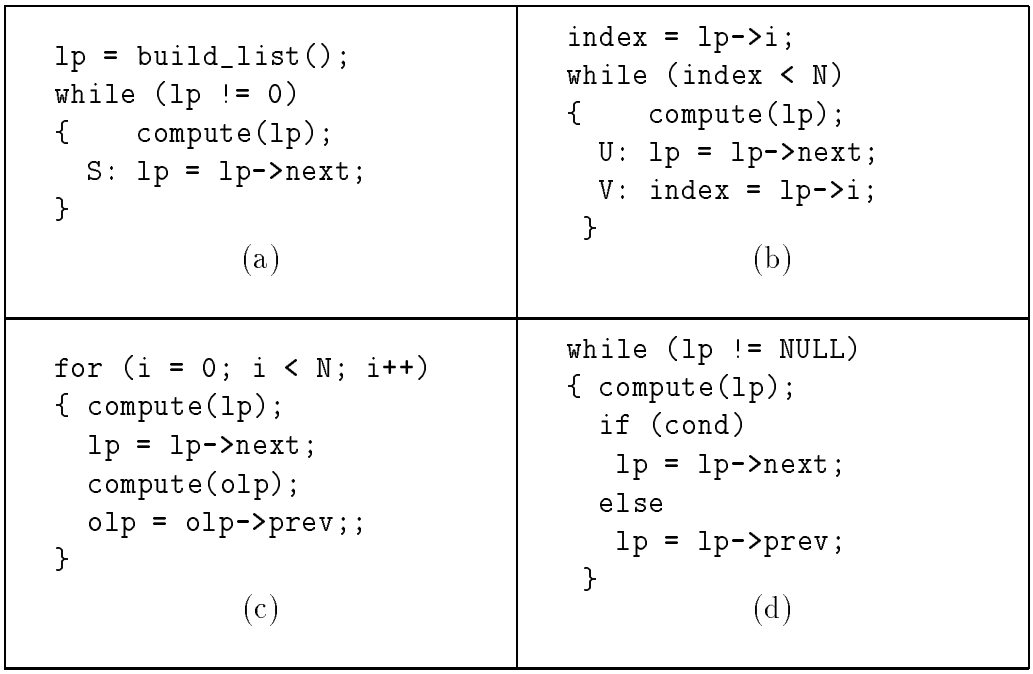


Figure 5.24: Identifying the Navigator

From the above examples, it can be noticed that the concepts of navigator and test/navigator expressions are analogous to the terms, loop index variable and induction variable, used in array dependence testing. Induction variables are expressed in terms of arithmetic expressions with respect to the loop index variable (`k = 2 * i + 5`). Test/navigator expressions are expressed in terms of field accesses with respect to the navigator.

In Figure 5.24(c) and (d), we show two loops for which the navigator cannot be identified (function *getLoopDefChain* returns NULL). The loop in Figure 5.24(c), uses two navigators (`lp` and `olp`). However, none of them is related to the loop test variable `i`. Hence, our algorithm cannot detect them. The loop in Figure 5.24(d) involves *conditional* definitions of the navigator. These definitions are not included in the definition chain, as they are not guaranteed to be executed in each loop iteration (conditions 2 and 3 for termination of the function *getLoopDefChains*). These examples show the limitations of our method. Our goal was to design simple and efficient techniques to detect the most frequently occuring cases. These techniques can be extended to include cases like above, but not without introducing considerable complexity.

5.5.2 Verifying the Navigator

Once a navigator is identified, and a navigator expression is constructed, the next step is to verify if the navigator visits a distinct node in the data structure in each iteration. Thus, the function *findNavigator* checks that none of the fields in the navigator expression (*navigator fields*) are updated inside the loop. For this purpose connection-based heap read/write sets are used as shown in Figure 5.18.

This check ensures that the navigator advances in a regular fashion from one iteration to the next iteration, i.e., the fields along which the data structure is navigated remain static through the loop execution. Further, note that the navigator is *definitely* advanced in each iteration using the navigator expression, and *not conditionally*. From this information, we can make the following important observations:

- If the shape attribute of the navigator is Tree or DAG, the navigator expression will lead the navigator to a distinct node in the data structure, in each iteration. In this case the data structure is acyclic, and since the navigator is advanced using the same expression every iteration (`lp = lp->next`), it cannot *revisit* a node.

- If the shape attribute of the navigator is Cycle, the above claim still holds in an important case. This case represents loops, where the loop test involves testing a heap-directed pointer, say `ptr`, against a constant `(ptr != NULL)` or another pointer that is loop-invariant `(ptr != b)`, where `b` could be the navigator for the outer loop. Such loops very frequently arise in C programs.

  For such loops, if the navigator (`ptr`) visits a given node a second time, the loop will execute infinitely. This is because the navigator fields are not updated inside the loop body. Consider again the loop in Figure 5.24(a). Suppose after three iterations, its navigator `lp` visits the node it visited at the beginning of the loop. Since the navigator field `next` is not updated, `lp` will visit this node every three iterations, and the condition `(lp == NULL)` will never be satisfied, giving an infinite loop. Note that during good loop detection, we have already ensured that the only exit point for the loop is the loop condition test.

  Thus, with the assumption that a loop does not run infinitely, we can infer that its navigator visits a distinct node in each iteration, for an important class of loops. These loops typically traverse parts of a cyclic data structure in an acyclic

fashion: for example a loop that traverses a doubly linked list only using the `next` link. We term such loops as *acyclic loops*. Note that programs can contain infinite loops, and we apply parallelizing transformations on *acyclic* loops only if the programmer guarantees the absence of infinite loops via an assertion.

Finally, if we are not able to find a valid navigator for the loop, the loop is flagged as non-*foreach* and is not analyzed further.

5.5.3 Detecting Heap-based Loop-carried Dependences

Once a valid navigator is found for a loop, we check if the loop involves any LCDs with respect to heap accesses. The function *heapLCD* in Figure 5.25, outlines this dependence test. It takes as input any two statements belonging to the loop, and determines if an LCD of depType (flow, anti or output) exists between them with respect to heap accesses. The test is invoked only for statements which are reported to be OnlyHeap dependent by the *stackTest*, and dependent by the connection analysis test *connectionTest*. This implies that they access the same field from nodes belonging to the same heap data structure, and can induce LCDs.

The test *heapLCD* works as follows. Given two statements from the loop, *stmtS* and *stmtT*, and the pointers they use to access the heap (*ptrS* and *ptrT*), the test first constructs the access paths for the two pointers with respect to the navigator. This is similar to constructing the access paths with respect to the anchor, for the *shapeTest*. Next, the *heapLCD* test compares the two access paths to detect if they can introduce an LCD. These access paths are termed as *navigator access paths* (NAPs). In case, the NAPs cannot be constructed, dependence is reported.

Before, further describing the *heapLCD* test, we first illustrate some situations where heap-based LCDs arise. While traversing a heap data structure, an LCD can be introduced when fields of *neighbor* nodes, i.e., nodes other than the one being currently visited by the navigator, are accessed. Consider for example, a loop traversing a list doing the following assignment `lp->next->i = lp->i`. One iteration writes to the field of a node, that will be visited by the navigator in the next iteration. Another situation where an LCD can arise is when two or more iterations access the same node in the data structure. For example, if all nodes in the list traversed by `lp` contain a

```
/* check for heap-based loop-carried dependencies */
fun heapLCD(stmtS, stmtT, loopStmt, depType) =
  [pSetS, pSetT] = heapAccessPtrs(stmtS, stmtT, depType);
  navigator = loopStmt.navigator;
  foreach pair (ptrS, ptrT) ∈ pSetS×pSetT
    [defS, defT] = getLoopDefChains(ptrS, stmtS, ptrT, stmtT, loopStmt);
    [pathS, pathT] = getPathExprs(navigator, defS, defT);
    if (!pathS or !pathT)
      return DEP; /* reference cannot be expressed in terms of  navigator */
    if (isTree(navigator.shape)) /* traversing a tree-like structure */
      if (fieldsUpdated(pathS, loopStmt, navigator) ||
          fieldsUpdated(pathT, loopStmt, navigator))
        return DEP; /* structural modification involved */
      if (navigatorFieldsUsed(pathS, pathT, loopStmt.navigatorExpr))
        return DEP; /* can access nodes from other iterations */
    else if (isAcyclicLoop(loopStmt)) /* loop test is (ptr != someConstant) */
      if (ptrFieldsUsed(pathS, stmtS, pathT, stmtT))
        return DEP; /* can access nodes from other iterations */
    else return DEP; /* cannot check this depenence: assume dependence */
  return NODEP; /* no loop-carried dependence detected */
```

Figure 5.25: Test for Loop-carried Heap Dependences

pointer to a header node and update its `num` field when visited (`lp->hdr->num++`), we have an LCD from one iteration to the next.

To access the neighbor nodes via the navigator, *pointer fields* must be traversed. Thus a NAP can lead to a neighbor node only if it involves one or more pointer fields. With this observation, we can infer that two statements can induce an LCD, only if one of the NAPs involves pointer fields. Otherwise, the NAPs lead to fields in the node currently being visited by the navigator, without introducing any LCD.

The function *heapLCD* essentially makes the above check. Additionally, it makes a weaker check if the shape attribute of the anchor is Tree. In this case, the NAPs can use pointer fields other than the navigator fields. For example, for the above mentioned list traversal, if the shape attribute of `lp` is Tree, the statement `lp->hdr->num++` will not induce an LCD. This is because the header node cannot be common for any two nodes in the list, else it will violate the Tree shape attribute. However, the assignment statement `lp->next->i = lp->i` will still induce the dependence as the access path `lp->next->i` uses the pointer field `next`, which is a navigator field.

We further illustrate the interaction between shape attribute and heap-based LCDs in Figure 5.26. Part (a) of the figure shows a loop traversing a linked list from the *graphics* benchmark described in Table 3.2 (page number 70). The navigator for the loop is the pointer `index` and the navigator expression is `index->f`. The loop however updates fields of the neighbor nodes accessed via the NAP `index->g`.

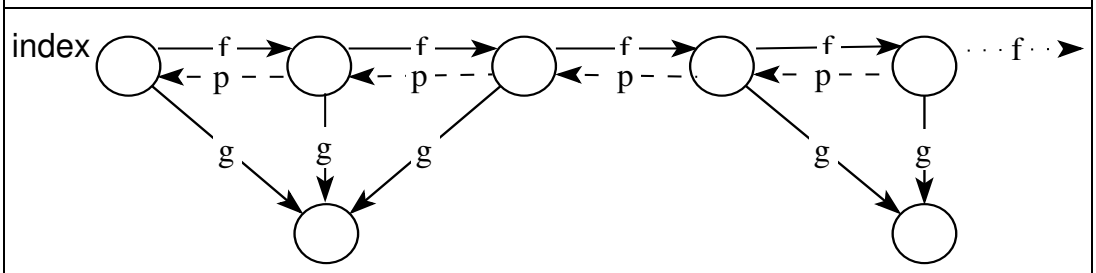
If the shape attribute of the navigator (`index`) is DAG or Cycle, we can have the situation as shown in Figure 5.26(b). Here, every three nodes in the list, *share* the neighbor node accessible via the field `g`. Thus the write heap accesses of the form `(*temp_278).tx`, with NAP for the pointer `temp_278` as `index->g`, will write to the same location for every group of three iterations, thereby inducing LCDs. However, if the shape attribute of the navigator (`index`) is Tree, the scenario will be as shown in Figure 5.26(c). Here, the neighbor nodes accessible via the field `g`, cannot be shared between any two nodes of the list. Thus heap accesses with NAPs of the form `index->g`, must lead to a distinct node every iteration, and cannot induce LCDs.

Based on the above discussion, the *heapLCD* test determines if two given NAPs induce an LCD, using the following checks:

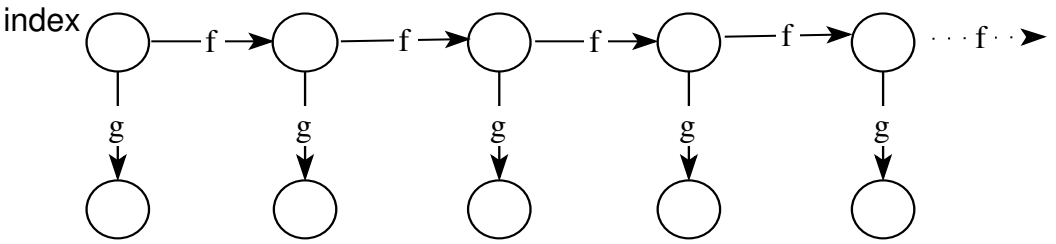
1. If the shape attribute of the navigator is Tree, and the NAPs do not use *navigator fields*, and do not update *pointer fields*; the statements are assumed to

```
index = (*o).Points;
while ( (index != 0) )
{
  temp_277 = (*index).g;
  temp_276 = (*temp_277);
  hp = TPointToHPoint(temp_276);
  hp = MultMatrixHPoint(Rot, hp);
  temp_278 = (*index).g;
  (*temp_278).tx = hp.x;
  (*temp_278).ty = hp.y;
  (*temp_278).tz = hp.z;
  index = (*index).f;
}
```

(a) Example Loop



(b) With DAG/Cycle Shape Attribute



(c) With Tree Shape Attribute

Figure 5.26: Loop-carried Dependencies and Shape Attribute

not induce an LCD. The last condition is required to ensure that the shape attribute of the navigator remains as Tree, throughout the loop.

2. If the the shape attribute of the navigator is DAG or Cycle, but the loop is *acyclic*, and the NAPs do not use *pointer fields*; the statements are assumed to *not* induce an LCD.

3. If none of the above conditions apply, an LCD is reported, the loop is flagged as non-*foreach*, and is not analyzed further.

5.5.4 Overall Strategy

Once we have detected a good loop with a navigator and with no heap-based LCDs, we flag this loop as a *foreach* loop with respect to the heap accesses. To identify it as a real *foreach* loop, we use existing tests implemented in the McCAT C compiler to check against LCDs induced by accesses to scalar variables, array references and stack-based indirect references.

To detect LCDs with respect to stack accesses (scalars and indirect references to the stack), the following check is performed: if the use (read access) of a stack location at statement `S` in the loop, is reached by a definition from statement `T` in the loop, and statement `T` lexically succeeds statement `S`, the statements potentially induce an LCD. For example, in the loop shown in Figure 5.28(a), the stack location `tmp.P` is used before being defined within the loop. The first iteration reads the value of `tmp.P` written by a definition outside the loop. However, the successive iterations read the value written by the previous iteration, thereby inducing an LCD. In the McCAT implementation, the stack read/write set information is used to compute the use/def sets for each stack access. These use/def sets are then used to detect LCDs as discussed above.

Once the set of LCDs with respect to stack accesses is computed, all LCDs with respect to the symbolic stack location *heap* are excluded from this set. These LCDs are already checked using the *connectionTest* and the *heapLCD* test. Further, the LCDs induced by accesses to the loop navigator, are also excluded. As discussed in section 5.1, a *foreach* loop intrinsically includes an LCD with respect to the navigator, and the parallelizing transformations for *foreach* loops (Figure 5.2 and Figure 5.3),

are designed to take this LCD into account. At this stage, if the set of LCDs is empty, the loop is considered *foreach* with respect to stack accesses.

To detect LCDs with respect to array references, special subscript tests are used. For example, array references `a[i]` and `a[j]` can induce an LCD if the subscript tests can detect that `j` can be expressed as an affine function of `i` (`j = m * i +/- k` where `m` and `k` denote constants). Further details can be found in [Jus94, Lap97]. If no array-based LCDs are detected, the loop is labeled as a real *foreach* loop, and can be accordingly transformed.

5.5.5 Identifying *forall* Loops

To identify heap-based *forall* loops of the type shown in Figure 5.1(b) (re-illustrated in Figure 5.27), we use a similar strategy as for detecting *foreach* loops. The key difference is that the navigator for the *forall* loops is an integer, and the navigator expression is an integer expression. For example, for the loop in Figure 5.27, the navigator is the variable `i` and the navigator expression is the variable `i + 1`.

```
for (i = 0; i < N; i++)
{
  t = list_arr[i];
  foo(t, x, y);
}
```

Figure 5.27: Example *forall* Loop

The tests for checking heap-based LCDs are modified to compare access paths which consist of array expressions as opposed to pointer references. To this end subscript tests developed for array dependence testing are used. For example, in Figure 5.27, the navigator access path for the heap pointer `t` is computed as `list_arr[i]`. From the information that `i` is a navigator, and the shape attribute of `list_arr` is Tree, the test infers that pointer `t` accesses a disjoint list in each iteration and cannot induce an LCD. Further, if it needs to compare access paths of the form `list_arr[i + j]` and `list_arr[i + k]`, it uses subscript tests from array dependence testing [Jus94, Lap97], to identify LCDs.

Program	Description	Principal Data Structures
treeadd	Tree Addition	Binary Tree
power	Power System Optimization	k-ary Tree
circuit	Sparse Matrix Solver	Doubly-linked Lists
pug	Grid Triangulation	Interconnected Linked Lists

Table 5.1: Benchmark Descriptions

5.6 Experimental Results

We have implemented the dependence tests described in sections 5.4 and 5.5, in the framework of our McCAT C compiler (Figure 2.1, page number 27). We have studied their effectiveness with respect to a set of four recursive data structure based C benchmark programs described in Table 5.1. Two of these benchmarks (*power* and *circuit*), are the same as described in Table 3.2 (page number 70). The results are summarized in Table 5.2. We discuss the results for each benchmark in detail below.

The benchmark *treeadd* implements the tree addition function shown in Figure 5.1(a) (page number 121). For this benchmark, we are able to find two pairs of parallel function calls. The first pair includes the two calls to the function `buildTree`, which respectively build the left and right sub-trees in a recursive fashion. These two calls are identified as parallel by the connection dependence test, as they always access disjoint data structures (which are newly allocated). The second pair includes the two calls to the function `treeAdd` as shown in Figure 5.1(a). To identify these calls as parallel, shape dependence test is used as they access disjoint sub-pieces of the same data structure.

The benchmark *power* implements a power network tree [RCRH95]. Each node uses an array of pointers to point to its child nodes, as opposed to named links like `left` and `right`. Shape analysis identifies the shape of this data structure as Tree. Using our tests for identifying heap-based loop carried dependences, we find four *forall* loops in this benchmark, with two different patterns. In Figure 5.28(a) and Figure 5.28(c), we show one loop representing each pattern.

For the loop in Figure 5.28(a), the absence of heap-based LCDs is detected by using shape and navigator information. Here `temp_23` is an array of pointers, with

Program	Parallel Calls	*foreach* Loops	*forall* Loops
treeadd	4	0	0
power	0	0	4
circuit	0	14	0
pug	0	5	0

Table 5.2: Benchmark Results

each pointer pointing to a disjoint tree data structure. Its shape attribute is identified as Tree. The navigator for this loop is the index `i`. Thus this loop is similar to the loop shown in Figure 5.1(b), and is identified as a *forall* loop with respect to heap accesses. However, it contains LCDs due to the stack-based accesses `tmp.P` and `tmp.Q`, which are involved in reduction operations. Since compilers are able to parallelize reduction operations, the above loop can be parallelized using these techniques.

The loop in Figure 5.28(c) recursively builds part of power network tree. Each call to `build_lateral` allocates a new data structure, which is connected to the existing tree by the assignment `temp_21[i] = l`. Thus the two statements in the loop access disjoint data structures. In this case connection dependence test suffices to determine the absence of LCDs and the loop is identified as *forall*.

The benchmark *pug* triangulates an unstructured grid and uses a complex cyclic structure, which includes several lists linked together via various pointers. Thus shape information is not useful for this benchmark. However, in its computation phase it contains several *acyclic loops*, which traverse one list at a time using a given link field. Additionally, these loops operate on only fields of the list node currently being visited, thus excluding LCDs. Our dependence tests find five *foreach* loops for this benchmark, which include the two most compute-intensive loops. One such loop is shown in Figure 5.28(b). In this loop, majority of the heap accesses are read accesses, with only two write access through the array references `temp_2598[0]` and `temp_2598[1]`. However, this array is embedded inside the list node being visited, so it cannot be shared with other nodes, thus excluding the possibility of LCDs.

The benchmark *circuit* implements a sparse matrix used in circuit simulation [HHN94]. The matrix is implemented using doubly linked lists, with nodes having links to nodes in the previous and next rows as well as columns. Again shape information is not useful for this benchmark. However, like the *pug* benchmark, it does majority of its

```
temp_23 = (*r).feeders;
/* heap forall loop */
for (i = 0; i <= 9; i = i + 1)
{
  l = temp_23[i];
  theta_R = (*r).theta_R;
  theta_I = (*r).theta_I;
  a = Compute_Lateral(l, theta_R,
       theta_I, theta_R, theta_I);
  temp_24 = tmp.P;
  temp_25 = a.P;
  tmp.P = (temp_24 + temp_25);
  temp_26 = tmp.Q;
  temp_27 = a.Q;
  tmp.Q = (temp_26 + temp_27);
}
```

(a) power

```
/* foreach loop */
 while (n != 0)
 {
   temp_2598 = (*n).phi;
   temp_2601 = (*n).x;
   temp_2600 = (temp_2601 *
                temp_2601);
   temp_2599 = (5.0 * temp_2600);
   temp_2605 = (*n).y;
   temp_2604 = (temp_2605 *
                temp_2605);
   temp_2603 = (6.0 * temp_2604);
   temp_2598[0] = (temp_2599 -
                   temp_2603);
   temp_2598[1] = 0.0;
   n = (*n).next;
 }
```

(b) pug

```
 temp_21 = (*b).leaves;
 /* forall loop */
for (i = 0; i <= 9; i = i + 1)
{
  l = build_lateral();
  temp_21[i] = l;
}
```

(c) power

```
/* foreach loop */
 while (e != 0)
{
  temp_327 = (*e).markowitzProduct;
  temp_328 = (int ) count;
  temp_329 = (temp_328 - 1);
  temp_330 = (long ) temp_329;
  (*e).markowitzProduct =
           (temp_327 * temp_330);
  e = (*e).nrowE;
}
```

(d) circuit

Figure 5.28: *forall* and *foreach* Loops from Benchmark Codes

computation in *acyclic loops*, traversing lists representing a particular row or column of the matrix. Our tests detect fourteen *foreach* loops for this benchmark with respect to heap accesses, seven of which involve reduction operations on scalar variables, and the remaining seven are true *foreach* loops, with only the navigator involving an LCD. A sample *foreach* loop from this benchmark is shown in Figure 5.28(d), which also represents the most compute-intensive part of the code.

Finally, based on our experience with hand-written Earth-C [HTZ+96] versions of the benchmarks *treeadd* and *power*, the parallelism detected by our dependence tests, respectively provides speed-up by factors of 16 and 12 on the EARTH-MANNA multithreaded machine [HMT+95] using 16 processors (Figure 4.11, page number 103).

5.7 Related Work

As summarized in section 1.2.2 in chapter 1, a considerable amount of work has been done on analyzing programs with recursive heap data structures [JM82, CWZ90, HN90, PCK93, Deu94, GH95, GH96a, GH96b, SRW96]. These analyses either explicitly estimate the structure of heap as a storage shape graph [CWZ90], or compute some abstract relationships between heap-directed pointers [HN90].

More directly related to this chapter are methods that use the results of heap pointer analysis in the context of dependence analysis and parallelization. These approaches include: techniques using path expressions to name locations [LH88], using syntax trees to name locations [Gua88], extending k-limited graphs with location names[HPR89]; and dependence analysis based on shape information and path expressions [HN90] The focus of these techniques is on identifying function-call parallelism for recursive data structures, and the heap analyses used are substantially more complex than our connection and shape analyses. Further, they do not consider the detection of loop parallelism. Finally, none of these techniques consider the presence of stack-directed pointers.

In contrast to the above techniques which are based on automatic heap analysis, Hummel et al. [HHN94] use a language-based approach. They rely on the programmer to provide the information about the shape of the data structure via aliasing axioms. To compute dependence between two statements, they collect access paths with respect to an anchor. A theorem prover is used to identify if, given the aliasing

axioms, the access paths can lead to the same node. This approach is quite powerful, as the aliasing axioms can accurately express the shape of even complex cyclic data structures, and the theorem prover can compare complex access paths.

In our dependence tests, we also use the concept of collecting access paths with respect to a common anchor. However, we rely on connection and shape information that is automatically computed, focus on identifying specific parallelism patterns, and thus use only two types of access path comparisons (equivPaths and subPath[4]), which can be done efficiently. Further, we also present techniques to identify loop-carried dependences with respect to accesses to recursive data structures, which are crucial for detecting loop parallelism. To our knowledge, previous work on dependence analysis for recursive data structures, has not addressed the problem of detecting loop-carried dependencies.

5.8 Summary

This chapter has focused on how to use pointer analysis for detecting parallelism. We used a family of pointer analyses that run in a hierarchical fashion, and designed separate dependence tests for each analysis. We briefly described how to use points-to based stack read/write sets and connection-based heap read/write sets, for dependence testing. For using shape information, we presented special dependence tests that build access paths with respect to a common anchor, and then compare the two paths in view of the shape information available. Further, we introduced a separate test for identifying loop-carried dependences, which is crucial for detecting loop-parallelism.

We also proposed the *foreach* loop construct for pointer-chasing loops, along with techniques to extract the parallelism available. Finally, we presented experimental results for a set of C benchmark programs, demonstrating that the parallelism patterns identified in section 5.1 frequently arise in recursive data structure based C programs, and that our dependence tests can effectively detect such parallelism.

[4]The disjointPath path comparison is done via a combination of equivPaths and subPath comparisons.

Chapter 6

Conclusions and Future Work

In this thesis we have presented effective techniques for putting pointer analysis to work, and provided detailed experimental evaluation of the benefits achieved. The experiments were performed using the McCAT C compiler. We augmented the existing implementations of the McCAT pointer analyses to compute read/write sets, and implemented the various compiler optimizations to evaluate their effectiveness. Below we briefly summarize the main contributions and the salient points of this thesis.

We first summarized the state-of-the-art techniques for pointer analysis, and categorized them as store-based and storeless approaches. Store-based techniques attempt to identify possible targets of a given pointer, while storeless techniques estimate abstract relationships between pointers themselves. We demonstrated that store-based approaches are directly suited to analyzing stack-directed pointers, as their targets have compile-time names. For analyzing heap-directed pointers, we proposed using storeless analyses, as locations in the heap are inherently anonymous and do not have natural compile-time names. Following this approach, we used the family of pointer analyses implemented in the McCAT compiler: store-based points-to analysis [EGH94] for stack-directed pointers, and two storeless heap analyses, connection analysis [GH96a] and shape analysis [GH96b] for heap-directed pointers. We used these analyses to study the applications and benefits of pointer analysis.

We illustrated how the presence of pointers considerably inhibits the power of compiler optimizations (section 1.4.1), and we then demonstrated that the fundamental problem is accurate computation of read/write sets for statements involving

pointer indirections and function calls using pointer arguments. In the absence of pointer analysis information, the compiler has to assume that a pointer dereference can access any memory location, which results in very conservative read/write sets. We briefly described how one can compute accurate read/write sets from points-to information (section 2.5). The computation was shown to be straightforward for statements involving indirection: it simply requires identifying the potential set of targets for the dereferenced pointer, and resolving the indirect reference into a set of named memory locations. For computing the sets for function calls, the invocation graph of the program is traversed.

For a storeless heap analysis like connection analysis, we showed that computing read/write sets is not so obvious (section 3.2). Connection analysis provides highly accurate information at a given program point, but does not provide a mechanism to relate information across two different program points, which is the goal of read/write set computation. To overcome this problem, we introduced the concept of anchor handles in connection analysis (section 3.3). Anchor handles are analysis-generated symbolic names that capture connection relationships with respect to specific incarnations of pointers at program points of interest. We augmented connection analysis with ghost copy assignments at appropriate program points to include anchor handles. We also discussed how to compute read/write sets for heap locations in terms of anchored locations, described our efficient implementation of anchor-augmented connection analysis in the McCAT C compiler (section 3.4), and demonstrated its efficiency via empirical results from a set of 16 pointer-intensive C benchmark programs.

Based on both the points-to based stack read/write sets and the connection-based heap read/write sets, we demonstrated a variety of applications of pointer analysis (chapter 4). We described three important scalar optimizations, loop invariant removal, location invariant removal and common subexpression elimination, that can include optimization of computations using pointers (section 4.1). We provided extensive static and dynamic measurements, including measuring runtime improvement achieved due to these scalar optimizations, for a set of 12 pointer-intensive C benchmarks (section 4.2). We showed that with our sharper read/write set information, these optimizations provide additional runtime speed-up of up to 10%, resulting from up to 35% reduction in the number of memory references made. These improvements are measured over a state-of-the-art optimizing compiler (gcc compiling at the highest level of optimization), and hence are quite significant, specially in the context of the above scalar optimizations. In the context of parallel EARTH-C programs,

we reported that the scalar optimizations provide up to 25% additional speedup by eliminating up to 98% remote memory operations (section 4.3).

We also examined the effect of accurate read/write sets on array dependence testers (section 4.4). In C programs, the majority of the array references are pointer-based, as arrays are either dynamically-allocated or passed as pointer arguments to functions. An array dependence tester (ADT) tries to establish that two references to a given array access disjoint memory locations, using a variety of subscript tests. However it first needs to identify if the given array references access the same array. We showed that in the presence of pointer-based array references, it is non-trivial to obtain such information, and in its absence the ADT can report many spurious dependences. With our read/write sets, this information can be easily obtained. We presented empirical results in Table 4.6, which indicate that the ADT is able to reduce array dependences from 138 to 46 for the *ear* benchmark, and from 70 to 7 for the *nrcode2* benchmark, thereby increasing the number of forall loops respectively from 9 to 19 and from 3 to 6. These improvements can produce significant performance gains for parallelized versions of the given programs.

Besides enabling more aggressive code optimization, we also developed a tool to use our read/write set information for program understanding/debugging purposes (section 4.5). The tool interfaces with a Web browser, and can be used to browse summary read/write set information, and the information collected by pointer analyses themselves, by simply selecting the program point(s) of interest. Being Web-based, this tool can be conveniently used from any site via the Internet. Finally, we also described how our shaper read/write sets can improve the effectiveness of more advanced compiler applications, including dynamic compilation, prefetching for recursive data structures, and identification of memory leaks (section 4.6).

Based on the pointer analysis information, we also developed effective techniques to detect parallelism in C programs with recursive data structures. We identified three commonly occuring parallel computation patterns in C programs, which include parallel recursive calls, and *forall* and *foreach* loops traversing recursive data structures (section 5.1). We designed a hierarchy of dependence tests to automatically detect these patterns. This hierarchy uses information from each pointer analysis via a separate dependence test.

We briefly described how to use points-to based stack read/write sets and connection-based heap read/write sets, for dependence testing. For using shape information, we

presented special dependence tests that build access paths with respect to a common anchor, and then compare the two paths in view of the shape information available. Further, we introduced a separate test for identifying loop-carried dependences, which is crucial for detecting loop-parallelism. We also proposed the *foreach* loop construct for pointer-chasing loops, along with techniques to extract the parallelism available. Finally, we presented experimental results for a set of C benchmark programs, demonstrating that our target parallelism patterns frequently arise in recursive data structure based C programs, and that our dependence tests can effectively detect such parallelism.

Above, we have summarized the main contributions of this thesis with its focus on "putting pointer analysis to work". In the light of our overall results, we believe we have shown that pointer analysis is an important part of an optimizing/parallelizing C compiler, and that significant benefits can be achieved from such an analysis, in the context of a variety of compiler applications.

6.1 Future Work

There are three major directions for future work. First, the benefits of our stack and heap read/write sets for low-level optimizations like register allocation and instruction scheduling, need to be evaluated. The three high-level scalar optimizations studied in this thesis, indirectly benefit the low-level optimizations, by replacing memory references involving pointer indirections, address-exposed variables and global variables, with scalar accesses. Via source-to-source transformation, the scalar replacement serves as an indirect mechanism to communicate the read/write set information to the low-level optimizer of the GNU C compiler.

However, if the optimizer has direct access to the read/write set information, it can perform more effective optimizations. For example, it can make better decisions about which variables to enregister, and build a more accurate dependence graph for instruction scheduling. Further, it can also take into account the interaction between various optimizations. This is important as optimizations can sometimes interact negatively. One such example was the negative interaction between CSE optimization and register allocation, for the *power* benchmark in our study (section 4.2.3).

Another area for further exploration is comparing the benefits of context-sensitive,

flow-sensitive pointer analyses (used in this thesis), vs. context-insensitive and flow-insensitive analyses [Ste96, SH97b]. The flow-insensitive analyses have almost linear time complexity in the size of the program, and can efficiently analyze large programs. Being flow- and context-insensitive, they provide less accurate information about pointer relationships as compared to context-sensitive analyses.

However, the flow-insensitive information may still suffice for many compiler optimizations. For example, to identify if an address-exposed variable is not accessed via a pointer indirection inside a loop, flow-insensitive information may suffice. Further insight can be gained via an experimental framework that computes read/write sets with respect to a flow-insensitive analysis, and uses them for applying various compiler optimizations. The number of optimizations applied and the actual performance improvement achieved, can then be compared with the case using read/write sets based on context-sensitive analyses. With this comparison, one can identify how much the additional accuracy of context-sensitive analyses actually helps the optimizations, and if it is worth its additional cost.

Besides comparing their relative benefits, developing a pointer analysis framework that uses the two approaches in a hierarchical fashion is another important area for further research. The main idea is to first apply the flow-insensitive analysis, and then use its results to partition the program into components for which context sensitive analyses can be conducted in isolation, and to identify which of these components require the more accurate context-sensitive analyses. This approach has been explored to some extent in [ZRL96, Ruf97].

In the context of using pointer analysis information for parallelism detection, there is considerable scope for future work. More general approaches need to be developed for extracting loop parallelism. For example, for *foreach* detection, our algorithms consider loops with only one navigator, that definitely advances in each iteration using a given navigator expression. More general approaches can include loops with multiple navigators, which can use a navigator expression from a set of expressions to advance to the next iteration.

In this thesis, we have discussed the notion of loop-carried dependence for pointer-based loops. A natural extension would be to add the concept of dependence distance in terms of the number of field accesses required. With dependence distance information, software pipelining can also be applied to pointer-based loops.

Finally, more powerful shape analyses need to be developed to accurately estimate

the shape properties of cyclic data structures with regular properties like doubly-linked lists and trees with parent pointers. One possibility is to abstract the shape of the data structure with respect to specific links. For example, the shape of a tree with parent pointers, can be abstracted as Tree with respect to the `left` and `right` links, and Cycle with respect to the `left` and `parent`, or the `right` and `parent` links.

Bibliography

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, May 1994. DIKU report 94/19.

[APC+96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 22–24, 1996. *SIGPLAN Notices,* 31(6), June 1996.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, corrected edition, 1988.

[Ban79] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer, 1988.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21:724–736, 1978.

[BGSP94] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 704–709, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, Texas, April 1992. Published as Rice COMP TR92-183.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[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.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[CL97] Keith D. Cooper and John Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, Las Vegas, Nevada, June 15–18, 1997. *SIGPLAN Notices,* 32(6), June 1997.

[Coo85] Keith D. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT and SIGPLAN.

[Cou86] D. Coutant. Retargetable high-level alias analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[CR82] Anita L. Chow and Andres Rudmik. The design of a data flow analyzer. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 106–113, Boston, Massachusetts, June 23–25, 1982. ACM SIGPLAN. *SIGPLAN Notices,* 17(6), June 1982.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the ACM 17th Symposium on Principles of Programming Languages*, pages 157–168, 1990.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, pages 2–13, April 1992.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[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] A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of IEEE 1994 International Conference on Computer Languages*, May 1994.

[Ema93] M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing compiler. Master's thesis, School of Computer Science, McGill University, September 1993.

[Ero94] A. M. Erosa. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, McGill University, May 1994.

[GH95] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Eight Workshop on Languages and Compilers for Parallel Computing*, number 1033 in Lecture Notes in Computer Science, pages 515–534, August 1995.

[GH96a] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *Intl. J. of Parallel Programming*, 24(6), pages 547–578, 1996.

[GH96b] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg, Florida, January 21–24, 1996.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Conf. Rec. of the 25th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, January 1998. To appear.

[Ghi92] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. ACAPS Technical Memo 62, School of Computer Science, McGill University, Montréal, Québec, December 1992. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[Ghi95] Rakesh Ghiya. Practical techniques for interprocedural heap analysis. Master's thesis, School of Computer Science, McGill University, May 1995.

[Gua88] V. A. Guarna Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 212–220, 1988.

[HDE+93] L. J. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth International Workshop on Languages and*

Compilers for Parallel Computing, volume 757 of *Lecture Notes in Computer Science*, pages 406–420. Springer-Verlag, 1993.

[HEGV93] L. J. Hendren, M. Emami, R. Ghiya, and C. Verbrugge. A practical context-sensitive interprocedural analysis framework for C compilers. ACAPS Technical Memo 72, School of Computer Science, McGill University, Montréal, Québec, July 1993.

[Hen90] L. J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, April 1990. TR 90-1114.

[HG92] L. J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the 4th IEEE International Conference on Computer Languages*, April 1992.

[HHN92a] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992.

[HHN92b] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[HHN94] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–229, June 1994.

[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.

[HN90] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Computing*, 1(1):35–47, January 1990.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 2nd edition, 1996.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[HS90] Ben Heggy and Mary Lou Soffa. Architectural support for register allocation in the presence of aliasing. In *Proceedings of Supercomputing 90*, 1990.

[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.

[Hud86] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986.

[ISY88] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM TOPLAS*, 10(4):555–578, October 1988.

[JM81] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications*, chapter 4, Flow Analysis and Optimization of LISP-like Structures, pages 102–131. Prentice-Hall, 1981.

[JM82] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Jus94] Justiani. An array dependence testing framework for the McCAT compiler. Master's thesis, School of Computer Science, McGill University, December 1994.

[KS93] N. Klarlund and M. Schwartzbach. Graph types. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[Lan92a] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), December 1992.

[Lan92b] William A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, 1992.

[Lap97] Christopher Lapkowski. A practical symbolic array dependence analysis framework for C. Master's thesis, School of Computer Science, McGill University, April 1997.

[Lar89] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, Berkeley, 1989.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[LH96] Christopher Lapkowski and Laurie J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In Proceedings of CASCON'96, Toronto, Ontario, November 1996.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review*, 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.

[LR92] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, California, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[MLR+93] T. Marlowe, W. Landi, B. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[Mye81] E. Myers. A precise interprocedural data flow algorithm. In *Proceedings of the ACM 8th Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[PCK93] John Plevyak, Andrew A. Chien, and Vijay Karamcheti. Analysis of dynamic structures for efficient parallel execution. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 37–56, Portland, Ore., August 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag. Published in 1994.

[PCK94] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Sixth International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 1994.

[RCRH95] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines.

ACM Transactions on Programming Languages and Systems, 17(2):233–263, March 1995.

[RM88] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, California, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[Ruf97] Erik Ruf. Partitioning dataflow analyses using types. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, Paris, France, January 15–17, 1997.

[SH97a] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 1997 Static Analysis Symposium*, Paris, France, Sep. 1997.

[SH97b] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 15–17, 1997.

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

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg, Florida, January 21–24, 1996.

[Sta92] Richard M. Stallman. *Using and Porting GNU CC.* Cambridge, Massachusetts, June 1992. Available via anonymous ftp from `prep.ai.mit.edu`.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, Florida, January 21–24, 1996.

[TGHG97] X. Tang, R. Ghiya, L. J. Hendren, and G. R. Gao. Heap analysis and optimizations for threaded programs. In *Proc. of the 1997 Conf. on Parallel Architectures and Compilation Techniques (PACT'97)*, San Francisco, Calif., Nov. 1997.

[Wei80] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[WH92] E. Wang and P. N. Hilfinger. Analysis of recursive types in LISP-like languages. In *Proceedings of the '92 ACM Conference on LISP and Functional Programming*, pages 216–225, June 1992.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[Wol89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, Massachusetts, 1989. In Research Monographs in Parallel and Distributed Computing; revised version of the author's Ph.D. dissertation published as Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering*, October 1996.