RICE UNIVERSITY

Interpolation Pointer Analysis for C

by

John Lu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Keith Cooper, Associate Professor
Department of Computer Science
Rice University

Linda Torczon, Faculty Fellow
Department of Computer Science
Rice University

Peter Druschel, Assistant Professor
Department of Computer Science
Rice University

J. Bartlett Sinclair, Associate Professor
Department of Electrical and Computer Engineering
Rice University

Houston, Texas

April, 1998

Interprocedural Pointer Analysis for C

John Lu

Abstract

Many powerful code optimization techniques rely on accurate information connecting the definitions and uses of values in a program. This information is difficult to produce for programs written with pointer-based languages such as C. For values in memory locations, accurate information is difficult to obtain at call sites and pointer-based memory operations. Most compilers conservatively assume that call sites and pointer-based operations may access any memory location. This greatly weakens the effectiveness of many optimizations and leaves opportunities for improvement.

This dissertation examines how to implement an interprocedural pointer analyzer for C in order to provide more accurate information about which memory locations call sites and pointer-based memory operations may access. The key issues of pointer analysis versus alias analysis, modeling of call sites, modeling of the heap, and usage of interprocedural path information are discussed. Solutions given for the first two questions and various solutions for the last two questions were used to build pointer analyzers of varying power and complexity. All versions of the analyzer were tested over a suite of programs, and we demonstrate that pointer analysis for programs of about 30K lines can easily be done with the computational power of current machines. The resulting pointer-analyzed programs were tested, and we show that their performance is better than non-analyzed programs. The data for each analyzers' performance and each analyzed programs' performance was used to look at the trade-offs among analysis time, analysis accuracy, and performance of the analyzed code. These results were also compared with interprocedural MOD/REF analysis. We also show that the analyzer can speed up the execution times of the rest our optimizer.

A new optimization, register promotion, that was specifically designed to use the information generated by pointer analysis was also developed. Register promotion moves a variable that normally resides in memory to a register for portions of the code in which it is safe to do so. Our experimental results for register promotion show that it is effective in reducing memory traffic.

Acknowledgments

First, I would like to thank my thesis advisor, Keith Cooper, for funding my research, and for reviewing and editing my work. I would also like to thank the other members of my thesis committee; Linda Torczon, Peter Druschel, and J. Bartlett Sinclair; for their time in reviewing my work and for their helpful advice. I would like to give special thanks to Linda Torczon for also funding my work through her Texas ATP award. I would like to thank my parents and my brother for their support in my graduate school years. Most of all I would like to thank God for His grace to persevere through obstacles in my research. I thank Him for directing me to a thesis topic that has been fruitful and interesting. I would also like to thank Him for providing numerous building blocks that have greatly aided my work. I would like to thank Tim Harvey and Preston Briggs for designing and supporting the compiler infrastructure on which this work was built. Their inclusion of the concept of "tags" in our infrastructure was extremely helpful for this work. The concept seems tailor made for use by pointer analysis. I would also like to thank Philip Schielke for building our C to `ILOC` front end (C2I) and for his addition of numerous changes to C2I so that my work could proceed. I would also like to thank Taylor Simpson for developing numerous optimization passes that utilized the "tag" information in our intermediate language. Without his passes it would not be as evident how much impact pointer analysis can have on subsequent optimization techniques. I would also like to thank Nathaniel McIntosh for his helpful advice and critiques throughout my graduate career. I would like to thank Gina Goff and Phillip Schielke for reviewing this thesis. I would like to thank Craig Johnson, Hope McIlwain, Blake Hurst, and Roger Huang for their encouragement, sympathy and prayers during the many difficult times in graduate school. I would also like to thank my fiancee, Yin Fern Ong, for her encouragement and reminders to finish my thesis.

I would like to thank Alejandro A. Schäffer for allowing us to test **mlink**. The program, **mlink**, comes from version 2.2 of the FASTLINK[22, 33] implementation of the LINKAGE[26] package for genetic linkage analysis. Genetic linkage analysis is a statistical technique used to map genes and find the approximate location of disease genes. LINKAGE/FASTLINK is a package of programs used by geneticists around the world to find the approximate location of disease genes. In our testing, **mlink** was run with the CLP data set. The CLP data set describes several families with many members who have autosomal dominant nonsyndromic cleft lip and palate[19]. I also thank Dr. Jacqueline T. Hecht for contributing disease family data. Development of the CLP data set was supported by grants from the National Institutes of Health and Shriners Hospital. Finally, I wish to thank Bill Landi for the **allroots** program.

Contents

Abstract ii
Acknowledgments iii
List of Illustrations viii

1 Introduction 1

2 Questions 3
2.1 Choice of problem 3
2.2 Modeling call sites 4
2.3 Program representation 11
2.4 Flow sensitivity 12
2.5 Modeling the heap 12
2.6 Context information 14
2.7 Summary 16

3 Related Work 18
3.1 Landi and Ryder 18
3.2 Choi, Burke and Carini 19
3.3 Emami, Ghiya and Hendren 19
3.4 Wilson and Lam 20
3.5 Ruf 22
3.6 Summary 24

4 *PL* Style Analysis 26
4.1 *PL* language 26
4.2 Pointer analysis 27
4.2.1 SSA form 28
4.2.2 Propagation 29
4.2.3 Running time 34
4.3 Modeling and analyzing C in *PL* 35
4.4 Implementation 43

4.4.1 C to ILOC 43
4.4.2 MOD/REF analysis 43
4.4.3 Propagation 46
4.4.4 Annotation 47
4.4.5 Other details 48
4.4.6 Path expressions 48
4.5 Sources of approximation 50
4.6 Summary 54

5 Results **55**
5.1 Analyzer performance 56
5.1.1 MOD/REF's impact on analysis 56
5.1.2 Multiple representations 58
5.1.3 Analyzer varying by heap model and path expressions 60
5.1.3.1 MEDIUM heap model 60
5.1.3.2 SMALL heap model 62
5.1.3.3 LARGE heap model 67
5.2 Analyzed code performance 78
5.2.1 No pointer analysis 79
5.2.2 MEDIUM heap model 79
5.2.3 SMALL heap model 86
5.2.4 LARGE heap model 90
5.2.5 Path expressions 93
5.2.6 Cache results 94
5.3 Optimizer performance 100

6 Register Promotion **102**
6.1 The algorithm 102
6.2 An example 105
6.3 Handling pointer-based references 107
6.4 Planned improvements 108
6.5 Results 109

7 Conclusion **119**

8 Future Work **120**
8.1 Heap modeling 120
8.2 Intraprocedural paths 121

8.3 Strong updates 121
8.4 Safety 121
8.5 Sources of approximation 122

A ILOC **124**

B Supplementary Results **129**
B.0.1 Cache Results 129

Bibliography **136**

Illustrations

2.1 Alias analysis vs. pointer analysis 5
2.2 Interactions between pointers and parameter binding 5
2.3 Representative names cause imprecision 7
2.4 Representative names cause incorrect results 8
2.5 Explicit names are precise 9
2.6 Explicit names produce correct results 10
2.7 Splitting the heap by call paths 13
2.8 False return problem 15
2.9 Pointer mixing 15
2.10 Path information improves MOD/REF analysis 17

3.1 Comparison of approaches 25

4.1 Operations with lists of modified and referenced regions 30
4.2 Before propagation step 32
4.3 After propagation step 33
4.4 Code for **PropagateToCall** 35
4.5 Code for **Propagate** 36
4.6 $POINTS\_TO$ set sizes 37
4.7 Program with one ALVR 39
4.8 Program with seven regions 40
4.9 **ADDR** operation is inadequate 41
4.10 Good and bad addressing calculation 42
4.11 C code translated to `ILOC` 44
4.12 MOD/REF analysis improves accuracy 47
4.13 Hand-made `ILOC` for an `atoi` function 48
4.14 Maintaining path expressions 49
4.15 ALVR causes approximation 53

5.1 Program descriptions 57

5.2 Experiment chart 58
5.3 Pointer analysis without MOD/REF analysis results 59
5.4 Comparison between multiple and single representations 61
5.5 Pointer analysis: baseline performance results 63
5.6 Pointer analysis: baseline performance results (cont.) 64
5.7 Pointer analysis: baseline pointer results 65
5.8 Pointer analysis: baseline pointer results (cont.) 66
5.9 SMALL heap model performance results 68
5.10 SMALL heap model performance results (cont.) 69
5.11 SMALL heap model pointer results 70
5.12 SMALL heap model pointer results (cont.) 71
5.13 SMALL/baseline heap model performance percentages 72
5.14 SMALL/baseline heap model performance percentages (cont.) 73
5.15 SMALL/baseline heap model pointer statistic percentages 74
5.16 SMALL/baseline heap model pointer statistic percentages (cont.) . . . 75
5.17 LARGE heap model performance results 76
5.18 LARGE heap model pointer results 76
5.19 LARGE/baseline heap model performance percentages 77
5.20 LARGE/baseline heap model pointer percentages 77
5.21 No pointer analysis: total operations 80
5.22 No pointer analysis: store operations 80
5.23 No pointer analysis: load operations 81
5.24 MEDIUM heap model: total operations 81
5.25 MEDIUM heap model: total operation removal percentages 82
5.26 MEDIUM heap model: stores 82
5.27 MEDIUM heap model: store removal percentages 83
5.28 MEDIUM heap model: loads 83
5.29 MEDIUM heap model: load removal percentages 84
5.30 How pointer analysis improves **water** 85
5.31 How pointer analysis improves **mlink**: source code 87
5.32 How pointer analysis improves **mlink**: machine code 88
5.33 SMALL heap model: total operations 88
5.34 SMALL heap model: stores 89
5.35 SMALL heap model: loads 89
5.36 Splitting the heap helps analysis 91
5.37 LARGE heap model: total operations 91
5.38 LARGE heap model: stores 92

5.39 LARGE heap model: loads 92
5.40 Summary of memory model impact on code performance 93
5.41 Path expressions help: example 1 94
5.42 Path expressions help: example 2 95
5.43 Normalized cache accesses 96
5.44 Single-level cache model: normalized misses 98
5.45 Two-level cache model: normalized misses (level one) 98
5.46 Two-level cache model: normalized misses (level two) 99
5.47 Optimization times (s): absolute 101
5.48 Optimization times: standardized (1.000 = original) 101

6.1 Equations for register promotion 103
6.2 An example 104
6.3 Promoting array references 106
6.4 Register promotion: total operations 110
6.5 Register promotion: total operations (cont.) 111
6.6 Register promotion: total operations (cont.) 112
6.7 Register promotion: store operations 113
6.8 Register promotion: store operations (cont.) 114
6.9 Register promotion: store operations (cont.) 115
6.10 Register promotion: load operations 116
6.11 Register promotion: load operations (cont.) 117
6.12 Register promotion: load operations (cont.) 118

8.1 Good and bad address creation 123
8.2 `ILOC` code with no address creation problems 123

A.1 Sample `ILOC` function 125
A.2 `ILOC` types 126
A.3 `ILOC` opcode roots 128

B.1 Supplementary results: total operations 130
B.2 Supplementary results: total operation removal percentages 130
B.3 Supplementary results: stores 131
B.4 Supplementary results: store removal percentages 131
B.5 Supplementary results: loads 132
B.6 Supplementary results: load removal percentages 132

B.7 Single-level cache statistics 133
B.8 Two-level cache statistics 134
B.9 Two-level cache statistics (cont.) 135

Chapter 1

Introduction

Many compiler optimization techniques rely on the availability of accurate information connecting the definitions and uses of values in a program. Examples of such transformations include dead code elimination[5], constant propagation[35], partial redundancy elimination[29], and global value numbering[34]. All of these optimizations propagate information along the connections between definitions and uses. In order for these optimizations to be effective, it is essential to make these connections as precise as possible. Unfortunately in C, pointer-based memory operations make it difficult to precisely determine the relationship between definitions and uses of values. Unless the compiler analyzes the behavior of pointer-based operations, it must assume that *any* pointer-based operation can access *any* memory location. Thus, any definition of a pointer-based value must be assumed to redefine every memory location and any use of a pointer-based value must be assumed to reference any memory location. This set of assumptions is usually much more conservative than necessary and frequently produces code that is slower than code that could be produced if pointer analysis were done.

This dissertation focuses on the design and empirical testing of a pointer analyzer for C. One of our main contributions is to examine a set of basic questions that any pointer analyzer for C must answer. We examine how previous work answered this set of questions. We also propose our own solutions to some of these questions and elucidate the trade-offs among the solutions. For some questions we explain why a particular solution is the best one. For other questions that have no clear answer, we test a variety of solutions in order to provide an empirical basis for making a decision.

Another contribution of this thesis is its extensive testing. We test our analyzer over a suite of fourteen programs which range in size from 200 to 28,000 lines of C code. We allow parameters for our analyzer to vary and record the impact of these changes on the performance statistics of our analyzer. We also record the impact of these changes on the performance of code analyzed by our analyzer. Whenever varying a parameter changes the performance of an analyzed program, we try to find the mechanism for the change. We look for an explicit example from the source code of our test suite that illustrates how the changed parameter can affect performance.

One of the strengths of our performance testing is our use of MOD/REF analysis as a benchmark for comparison. Like pointer analysis, MOD/REF analysis produces information about the memory locations that loads, stores, and calls can access. Unlike pointer analysis it is fast and simple, and it uses little memory. For these reasons, it is an excellent benchmark for comparison. We show that pointer analysis is useful by demonstrating that it improves the performance of code when compared against code produced with MOD/REF analysis.

In order to increase the usefulness of our pointer analysis we also develop a new optimization technique, register promotion, that is specifically designed to use the information produced by pointer analysis. Again, we examine the impact of this technique on the performance of analyzed code.

The rest of this dissertation is divided into seven chapters. In Chapter 2, we examine basic questions that any pointer analyzer must answer. For each question we examine various answers and describe the trade-offs among the answers. Chapter 3 discusses previous work in this area and how this work has answered these questions. We look at the research of five groups: Landi and Ryder; Choi, Burke, and Carini; Emami, Ghiya, and Hendren; Wilson and Lam; and Ruf. Chapter 4 presents an abstract language, *PL*, that has all the language features relevant to pointer analysis. We present a framework for doing pointer analysis on *PL* and a time bound for this analysis. We also show how a C program can have its pointer behavior modeled by a *PL* program. Chapter 5 contains performance results for the analyzer over a suite of fourteen programs. The impact of our analysis on the performance of the analyzed programs is also shown. Chapter 6 describes a transformation, register promotion, that utilizes the results of pointer analysis. Results for this transformation are also shown. In Chapter 7 we present our conclusions on pointer analysis based on the results of this thesis. Chapter 8 outlines possible areas of future research in pointer analysis.

Chapter 2

Questions

Many different approaches have been attempted to analyze the use of pointers in C. A useful way of categorizing pointer analysis is to consider the questions that must be answered in designing the analyzer. In this chapter we will examine six issues that are relevant to pointer analysis: alias pairs vs. points-to, modeling call sites, program representation, flow-sensitivity, heap model, and context information.

2.1 Choice of problem

The first design choice for an interprocedural pointer analyzer is: Will it analyze pointers or aliases? In alias analysis, the fundamental abstraction is the alias pair. An alias pair is simply a pair $\langle x, y \rangle$ where x and y are C expressions that may access the same memory location. The goal of alias analysis is to determine the set of alias pairs that can hold at a given point in a program. Using alias pairs is an unfortunate consequence of thinking of pointer analysis in terms of the framework used to analyze aliases in FORTRAN caused by call-by-reference parameters[13]. Formal parameters in a call-by-reference language are not equivalent to memory locations. They are alternative names given to memory locations within a function. In this situation, alias pairs are the correct information to gather.

In C, however, the situation is different. If we ignore unions,[1] each memory location has a unique name, the name with which it is declared. Thus, the distinction between a name and a memory location is both artificial and unnecessary in C. However, even though a memory location has a unique name in C, it can be accessed through other names—the names of pointers that point to it. We can derive a relationship, called "points-to," for a C program that embodies this information. The points-to relationship maps a variable name to a set of memory locations to which it may point. This more closely models the run-time situation in C; it appears to be the critical information to gather in analyzing the pointer behavior of C programs.

[1]When we have unions, we use the same name for each member of the union since they have overlapping memory locations.

Consider the solution that the two approaches yield for the fragment of code in Figure 2.1. After the second statement there are three valid alias relationships, but there are only two points-to relationships. The alias relationship, `<**pp,i>` can be derived from the other two alias relationships. If we read the alias relationship, $\langle *x, y \rangle$, as x points-to y, we find that the remaining two pieces of alias information and the points-to information are identical. The fact that removing redundant aliasing information gives us points-to information is an indication that points-to information is preferable.

An added benefit of adopting the points-to abstraction is that it handles function pointers in an easy and natural way. The same machinery used to analyze memory pointers can track pointers to functions, as long as the compiler can recognize both the creation of such pointers and their actual use. Hendren noted this as early as 1994 [16].

2.2 Modeling call sites

The analyzer must include a mechanism for modeling the effects of executing a procedure call and a subsequent return on the state of the program's name space. The complications that arise from these events are a result of the parameter binding mechanism in C, which relies on call-by-value parameters [2]. This seems simpler than the call-by-reference convention used in Fortran; however, the presence of pointer variables in C re-introduces all of the problems caused by reference parameters in other languages, as Figure 2.2 shows.

In designing an analyzer, this problem raises another critical question: How should the analyzer represent and name non-local memory? Two solutions have been proposed in the literature: using abstract but representative names and using explicit names.

In a framework that uses representative names, the analyzer creates a new abstract name for the non-local memory that can be referenced at each call site. These names are then "un-mapped" at the procedure's return to reflect the effect of the procedure call on any non-local memory that it may have accessed. The difficulty with using representative names is deciding how many representative names to create and how to associate these names with non-local memory. Depending on the choices made, representative names will either cause imprecision or incorrect results.

Consider Figure 2.3. To model the memory locations that can be accessed by parameters `*p1` and `*p2`, the analyzer could choose to create a single representative for each type of location. Thus one representative `r1` is created for the `*int` type, and another `r2` is created for the `int` type. Only a single name is used because the

Code	Alias Analysis	Pointer Analysis
`void f() {`		
`  int **pp,*p,i;`		
`  pp=&p;`	`<*pp,p>`	pp → $\boxed{\texttt{p}}$
`  p=&i;`	`<**pp,i> <*pp,p> <*p,i>`	pp → $\boxed{\texttt{p}}$ → $\boxed{\texttt{i}}$
`}`		

Figure 2.1 Alias analysis vs. pointer analysis

```
subroutine caller()
   integer a, b
   call called(a,b)
   end

subroutine called(r,s)
   r = 1
   s = 2
   end
```

Fortran version

```
void caller() {
   int a, b;
   called(&a,&b);
}

void called(int *r,int *s) {
   *r = 1;
   *s = 2;
}
```

C version

Figure 2.2 Interactions between pointers and parameter binding analyzer must assume that `*p1` and `*p2` may point to the same location. On exit from `called`, the analyzer will know that `r1` can point to `g1` and `g2`. When it maps these results back into the call site in `caller`, it will determine that `a` can point to either `g1` or `g2`. It will also discover that `b` can point to either `g1` or `g2`. The use of a single representative name has introduced imprecision into this simple example.

If we try to use multiple representatives for one type, this can lead to incorrect results. Consider the example in Figure 2.4. In this example we assume that every non-local location is distinct, and we give it a unique representative. Unfortunately, this causes representatives `r1` and `r3` to represent the same location. Since we use multiple representatives to represent one location, we cannot recognize that `*p` and `**pp` are aliases. Thus, we incorrectly conclude that `*p` has the value 4 at the return statement.

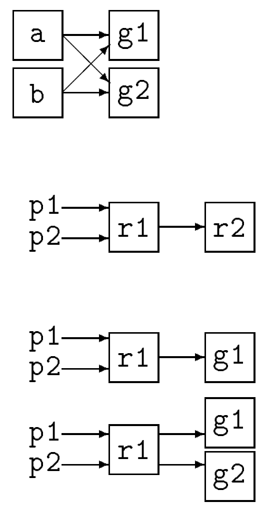
A further difficulty with representative names arises in handling arbitrarily-sized data-structures. The analyzer must decide how many names to create for each pointer; the only real hint available is the type of the pointer. Unfortunately, programmers can create types that are self-referential, like graphs. For a pointer to a graph, the analyzer cannot easily determine the number of representative names that must be created. This problem is analogous to modeling data structures in the heap.

These problems with using representative names can be eliminated by using explicit names. When the code passes a pointer into a function, the pointer can point to an explicitly-named object. In this case, the analyzer should use the explicit name of the object rather than a representative name. In the previous examples, using explicit names would produce different results.

Consider the case in which using representatives names gave imprecise results (see Figure 2.3). Figure 2.5, shows the results that would be obtained by using explicit names in this case. On entry to `called`, we precisely determine that `p1` points to `a` and `p2` points to `b`. This comes from looking at the parameters passed to `called`. Within `called`, we accurately set `a` to point to `g1` and `b` to point to `g2`. Since we use explicit names, this information does not need to be mapped back to corresponding names.

Using explicit names also gives us correct results for the example where representative names gave us incorrect results (see Figure 2.4). The results using explicit names for this example are shown in Figure 2.6. At the start to `called`, `*pp` and `p` are correctly shown as pointing to the same location. This is achieved by using the names passed in from the call site and preserving the points-to relationships that already exist among those names (*i.e.*, `ip` points to `i`). By using this information, we correctly determine that the return value `*p` equals 3.

```
void caller() {
  int *a,*b;

  called(&a,&b);
}


void called(int **p1, int **p2) {


  *p1=&g1;


  *p2=&g2;
}
```



$\boxed{\texttt{r1}}$ represents $\boxed{\texttt{a}}$ and $\boxed{\texttt{b}}$

Figure 2.3 Representative names cause imprecision

```
void caller() {
  int *ip,i;

  ip=&i;
  called(&i,&ip);
}
```

`int called(int *p,int **pp) {`	$p \rightarrow \boxed{\texttt{r1}}$ $pp \rightarrow \boxed{\texttt{r2}} \rightarrow \boxed{\texttt{r3}}$
`  *p=4;`	$\boxed{\texttt{r1}}=4$
`  **pp=3;`	$\boxed{\texttt{r3}}=3$
`  return(*p);`	$\boxed{\texttt{r1}}=4$
`}`	

$\boxed{\texttt{r1}}$ and $\boxed{\texttt{r3}}$ represent $\boxed{\texttt{i}}$

Figure 2.4 Representative names cause incorrect results

```
void caller() {
  int *a,*b;

  called(&a,&b);
}


void called(int **p1, int** p2) {


  *p1=&g1;


  *p2=&g2;
}
```

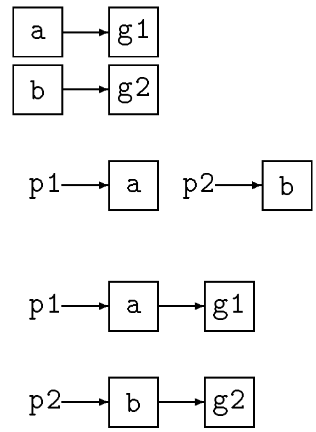


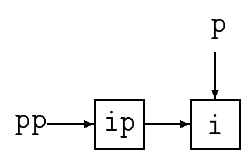
Figure 2.5 Explicit names are precise

Clearly, an analyzer based on explicit names will have its efficiency limited by the number of names that it must instantiate. Thus, from an implementation perspective and an efficiency perspective, it is important to limit the number of non-local variables that the analyzer believes a function can access.[2] Otherwise, using explicit names will become too costly. Non-local names arise from three sources: global variables, heap-based variables, and local variables whose addresses are explicitly computed. The number of globals is finite. Unfortunately, the number of heap-based variables is unpredictable, in the general case, as is the number of concurrently live instances of a specific local variable.

In using explicit names to handle procedure calls, care must be taken with addressed local variables of recursive functions (ALVRs). Since an ALVR can have more than one instance, the analyzer must be quite clear about what it means when it determines that an ALVR points to some name. Some approximation will be necessary. In our framework, we will approximate by having the single name for the ALVR represent all instantiations of the ALVR. If at a program point **p**, some instantiation of an ALVR, `r`, may point to a location `q`, then in our analysis at program point **p**,

[2]The analyzer must represent all of the names that are possible. What we would like to do is limit the number of spurious names.

```
void caller() {
  int *ip,i;

  ip=&i;
  called(&i,&ip);
}

int called(int *p,int **pp) {


  *p=4;


  **pp=3;


  return(*p);
}
```



$\boxed{\texttt{r1}}=4$

$\boxed{\texttt{r3}}=3$

$\boxed{\texttt{r1}}=3$

Figure 2.6 Explicit names produce correct results `r` should point to `q`. An important consequence of this approximation is that stores to an ALVR cannot "kill" old values. (*i.e.*, all pointer values in an ALVR before a store will still be in the ALVR after the store.) The non-addressed local variables of a recursive function need not be approximated in this way. When such a variable is accessed, only the most recent instance can be accessed. Thus, its name does not need to represent all instances, but only the topmost instantiation. See Sections 4.1, 4.2 and 4.3 for a more detailed discussion of this issue.

The choice between using explicit names or representative names may require trade-offs between space, speed, accuracy, and simplicity. Explicit names should be more accurate, but more space will probably be required to give a function access to all the non-local names it may use. The increased number of names used will require more time to analyze, but this is balanced by the fact that no mapping and unmapping functions are necessary. It is not clear which method will be faster. Section 5 presents performance results for our analyzer, which uses explicit names. We do not have comparable results for using representative names, because we did not implement it. Although using representative names may be faster, we felt that any implementation using representative names that was correct would be too imprecise. Since using explicit names is both correct and precise, and since it seems simpler (*e.g.*, it does not require complicated mapping functions), it seemed unnecessary to investigate representative names.

2.3 Program representation

Another issue that affects the design of both the analyzer and the analysis algorithm is the design of the intermediate form used to represent the program. In particular, the level of abstraction in the representation determines, to a large extent, both the difficulty of deriving the initial information and the ease with which the results can be applied to the resulting code.

- A more abstract, source-level representation can simplify gathering initial information; in a lower-level, assembly-like representation, some analysis may be required to recreate information that was obvious in the source code. For example, an address expression may be implicit at source-level but be distributed across several basic blocks at assembly-level.

- A more abstract, source-level representation imposes the programmer's name space onto the results of analysis; in an assembly-level representation, each individual subexpression may have a unique name. For example, a source-level analysis is unlikely to disambiguate two pointer references that use induction variables introduced by operator strength reduction [12].

In designing an analyzer, the compiler writer should consider the uses to which the results will be put. We intend to use the results of our analysis to drive optimizations on a low-level representation of the program. Therefore, our analyzer will work from an assembly-like representation of the code. In general, we would expect the results of analysis on the low-level code to be of greater interest to low-level optimization.

2.4 Flow sensitivity

In any interprocedural analyzer, the compiler writer must choose between a "flow-insensitive analysis" or a "flow-sensitive analysis". In general, the flow-insensitive algorithm will ignore control flow inside individual procedures, where the flow-sensitive algorithm will need to analyze the flow inside each procedure. For points-to information, the choice between flow insensitive and flow sensitive will change the results of the analysis significantly.[3]

Flow-sensitive analysis will, in practice, be slower [31]. We feel that the increased precision obtained with a flow-sensitive analysis is critical in computing points-to information. Thus, our analyzer uses a flow-sensitive algorithm.

2.5 Modeling the heap

A common use of pointers in C programs is to track the addresses of data structures that have been dynamically allocated on the heap. Thus, much effort described in the literature has been directed at discovering and representing the shape of pointer-based structures constructed in the heap [15, 10, 25, 20].

From our perspective, the issue of understanding the internal structure of the heap is orthogonal to the rest of the analyzer.

- The simplest solution available to us would model the whole heap with a single name. All loads and stores involving the heap would refer to this name. Since the name represents multiple locations, stores to it cannot kill old values.

- An easy improvement to the simple model would split the heap by call sites into the allocation routine. This would give a distinct name to the memory allocated by each call site, improving the precision of the analysis at a small cost in complexity, speed, and space.

- Further refinement in heap modeling is possible. It may be possible to split the heap memory allocated by one call site into separate regions, along either the

[3]There are problems for which this is not true. May Modifies information is the classic example of a problem that gains no additional information from considering intraprocedural control flow.

> time or space dimensions. For example, in the space dimension, we could split the memory generated by one call site based on the structure of the memory allocated at the call site. In the time dimension, we might try to generalize from splitting by call sites to splitting by call paths.

Splitting the heap by call paths can be helpful. For example, in Figure 2.7, the analyzer could return two separate names for the two calls to `My_Malloc`. However, for memory allocators that recycle memory, this will not be useful. In this case, from the point of view of our analyzer, the memory allocator behaves just like any other function that may return a pointer to the same memory location at different call sites (*e.g.*, a function that returns the start of a global linked-list). Thus we must return the same heap name for all calls to the allocator. A solution to this problem would be to allow the user to specify which calls allocate memory. This would accurately split the heap for all memory allocators.

Another popular approach to splitting the heap is *k-limiting*[21]. This approach allows up to *k* copies of a heap name to be generated where *k* is pre-selected. The goal of the *k-limited* approach is to allow for more accurate analysis of heap structures. If we have a heap-allocated linked-list, the first *k-1* nodes can be given unique names and the rest of the list can be approximated with the *k*-th name. This information is very difficult to generate. Furthermore, even if this information could be determined, it would not be very beneficial. In this analysis, a technique is useful only if it allows more precise determination of what memory locations loads and stores may access. Thus, splitting a heap name into two names, `@_heap1` and `@_heap2`, will be profitable only if memory operations that operate on only one of the two names can be found. Finding such operations should be easy to do when the heap is split by call sites.

```
main() {
  Node *a,*b;
  a=My_Malloc(sizeof(Node));
  b=My_Malloc(sizeof(Node));
}

Void *My_Malloc(Int size) {
  return malloc(size);
}
```

Figure 2.7 Splitting the heap by call paths

It may be very difficult when the heap is split by k-limiting. For example, we think it would be very rare to find a memory operation that operates on some subset of the first *k-1* elements of a linked-list. Besides being rare, if such a memory operation did exist, it would probably be very difficult for a compiler to determine that the operation only operated on a subset of the first *k-1* nodes.

2.6 Context information

Another choice facing the compiler writer is how much context to record and use during propagation. For each pointer, we can associate information with it that gives some idea about which calling-context generated it. This information can take the form of conditions that must be satisfied by a call site in order for a points-to relation to be valid. These conditions themselves are frequently points-to relations. A sample points-to relation with an associated condition might be `x` points to `y` if parameter `p` points to global `g` on entry to the procedure. Another form that has been used for context-information is to record the path (*i.e.*, list of call sites) through which a points-to relation was generated. There is a wide range of choices concerning how much path information to record. The simplest is to record no information; the most complicated is to record a full listing of all call sites through which a pointer was generated. Naturally, the simplest choice is also the least accurate as well as the quickest. The most complicated, full path information, can cause an exponential increase in the amount of time and space required to analyze a program. We may also choose to arbitrarily limit the number of call sites recorded, just as k-limiting places an arbitrary fixed bound on the length of models for heap-based data structures. See Figures 5.5, 5.6, 5.7, and 5.8 for our time and space requirements for performing pointer analysis with varying path lengths. Various other statistics are also shown.

Context information can improve pointer analysis in two ways. In Figure 2.8, if the analyzer lacks path information, it will conclude that `p1` can point to either `a1` or `a2`, and that `p2` points to the same values. This inaccuracy arises when the analyzer begins to analyze `called`. It begins by merging the points-to relations from two different contexts. In this case, it merges the fact that `p` can point to `a1` with the fact that `p` can point to `a2`. Propagating these facts through the return and back into the assignment in the calling procedure results in the erroneous (but conservative) points-to information. This problem has been termed the realizable path problem[23] or the false return problem[7].

In Figure 2.9, `pp` points to `p1` and `p2` and `p` points to `a1` and `a2`. There are no inaccuracies with this information, but without context information it is impossible to know that `pp` and `p` cannot simultaneously point to `p1` and `a2` respectively or `p2`

```
void caller1() {
  int a1,*p1;
  p1=called(&a1);
}

void caller2() {
  int a2,*p2;
  p2=called(&a2);
}

int *called(int *p) {
  return(p);
}
```

Figure 2.8 False return problem

```
void caller1() {
  int *p1,a1;
  called(&p1,&a1);
}

void caller2() {
  int *p2,a2;
  called(&p2,&a2);
}

void called(int **pp,int *p) {
  *pp=p;
}
```

Figure 2.9 Pointer mixing

and `a1` respectively. Thus, in `caller1` after the call to `called`, `p1` will needlessly point to `a2`. We have named this the pointer mixing problem.

Recent work by Ruf suggests that even with full path information, no increased accuracy is obtained for pointer analysis[32]. However, even though path information may not improve pointer analysis, its use may still be justified because it can improve the subsequent interprocedural MOD/REF analysis. Consider Figure 2.10. The store in `called` modifies `g1` and `g2`. Without path information we would have to summarize the effect of all calls to `called` as modifying both `g1` and `g2`. This is imprecise. With path information we can limit the modifications to a particular call site. This generates the more accurate information in the right column.

2.7 Summary

We have examined six questions relevant to pointer analysis in this chapter: alias pairs vs. points-to, modeling call sites, program representation, flow-sensitivity, heap model, and context information. Clear answers to the first two questions were developed (*i.e.*, use points-to analysis with explicit names). We have presented various possible solutions for how to model the heap and how to implement context information. Solutions for these problems that should increase the accuracy of pointer analysis were shown (*i.e.*, splitting the heap by call site and full path expressions). However, it is not clear if these solutions will have enough impact on the accuracy of pointer analysis and the speed of analyzed programs to justify their use over simpler solutions (*i.e.*, a single name for heap and no context information). The rest of this thesis investigates the benefits and costs of these solutions.

	MOD/REF Results	
	without path info	with path info
`void caller1() {`		
`  called(&g1);`	g1 g2	g1
`}`		
`void caller2() {`		
`  called(&g2);`	g1 g2	g2
`}`		
`void called(int *p) {`		
`  *p=3;`		
`}`		

Figure 2.10 Path information improves MOD/REF analysis

Chapter 3

Related Work

In this chapter we examine previous work on pointer analysis for C. This chapter is divided into six sections, one for each of five groups that has done work in this area. We present each group in the chronological order that their work was published. The five groups are: Landi and Ryder; Choi, Burke, and Carini; Emami, Ghiya, and Hendren; Wilson and Lam; and Ruf. In the last section we summarize previous work that has been done in this area.

3.1 Landi and Ryder

Landi and Ryder's work is one of the first attempts at alias analysis for C programs [23, 24]. Their view of this problem is clearly influenced by earlier work to analyze FORTRAN aliases. Because of this background, they focus on finding the set of possible aliases at every program point. They limit their work to a "C-like imperative programming language with sophisticated pointer usage and data structures, no type casting, explicit function calls, and arrays." In their work they treat arrays as aggregates. Names are *k-limited*[4] in order to ensure that only a finite number of names need to be analyzed. They calculate the set of aliases that may hold at a given program point, the *may-alias* problem, by first calculating the set of aliases that may hold on entry to a function if a given alias condition is satisfied, the *may-hold* problem. They show how the *may-alias* problem can easily be solved once the *may-hold* problem is solved. The solution to the *may-hold* problem incorporates some context information by associating an alias condition with the alias pair. For example, one condition may be that formal parameter `a` and location `*g` are aliases on entry to a function. A variable, whose name is hidden from a function, but which may be accessible through a pointer, is modeled with the name *non-visible*. This modeling is

[4]In this case, *k-limiting* limits names to a certain length. All names with greater length are represented by their longest prefix that is short enough to satisfy the *k-limiting* requirement. This is similar to the *k-limiting* done in analyzing heap structures, where arbitrary size structures are modeled by a finite number of nodes. In the heap analysis case, a finite number of nodes, *k-1*, is used to model a portion of the data structure. Subsequent nodes are all approximated by a single node.

not explained in detail. The heap is modeled by a name for each call site. Presumably these names are also *k-limited*. Landi and Ryder have implemented their analysis and have presented analysis times and accuracy results. They identify four base sources of approximation and prove that these are the only ones. Their first source of approximation is *k-limiting*; the others are due to lack of information about intraprocedural and interprocedural control flow.

3.2 Choi, Burke and Carini

Choi, Burke and Carini's research is another early work on alias analysis for C [11]. Like Landi and Ryder, their work is also based on alias pairs with names truncated by *k-limiting*. They note that only the *transitive reduction*[1] of the directed graph of alias relations needs to be kept.[5] They note that this more compact representation can also improve the accuracy of their analysis. Heap locations are named by the call path in which they are allocated. They also allow k copies of an object to be associated with each allocation site, and they divide their analysis into separate interprocedural and intraprocedural phases. Their interprocedural analysis has both flow-insensitive and flow-sensitive versions (context-insensitive and context-sensitive). Since pointers allow a callee to modify the aliasing relationships in its caller, their context-sensitive interprocedural analysis must be interleaved with their intra-procedural analysis. They achieve context-sensitivity by recording the last call site through which an alias pair passed. They handle invisible locations with representative names. In one example, they give an example where two pointer arguments of the same type are initialized to point at different representatives. This is not safe unless we guarantee that the two pointers cannot point to the same object. They do not give any performance data.

3.3 Emami, Ghiya and Hendren

The purpose of Emami, Ghiya, and Hendren's work was to provide the most context-sensitive analysis possible (see Section 2.6)[16]. They achieve this by performing their analysis over the invocation graph. The invocation graph is a directed tree of all possible call-stacks with a rooted node for the `main` function (*i.e.*, each possible call-stack is matched with a path in the invocation graph). Recursion is handled by terminating the graph when a function appears twice in a call-stack. The terminating

[5] The directed graph of alias relations has a node for each variable name in the program. A directed edge is placed from a variable to another variable if the variable at the source of the edge can be dereferenced to access the variable at the sink. Thus, for alias pair `<**a,b>` an edge would be placed from `a` to `b`.

node has an edge (this is not considered part of the graph) back to the node that started the recursion. Essentially, this is equivalent to using full path-information. They note that performing analysis over the invocation graph may require exponential time. They suggest that "sub-trees that have the same or similar invocation contexts" can be shared. This is the main idea behind Wilson and Lam's work (see Section 3.4).

This work is the first, we believe, to use the points-to abstraction rather than the alias pair abstraction. By using a points-to abstraction they can easily and naturally analyze function pointers. Analyzing function pointers is a necessity since they perform analysis over the whole invocation graph. This is accomplished by constructing the invocation graph while propagation is being done. When a function pointer is propagated to a call site, a new node is added to the call-graph corresponding to the target of this function pointer.

Emami, Ghiya, and Hendren use representative names (they call them symbolic names) to handle call sites. They note that two representative names should not be used to represent the same memory location. They also note that as many representatives as aliasing allows should be used in order to increase the accuracy of pointer analysis. They consider heap analysis an issue orthogonal to the rest of pointer analysis. Their focus is analyzing stack and global pointers. Thus, they simply model heap memory with one name. They note that pointers that reside in heap memory almost always point to heap memory. Thus, using a coarse model for heap memory will not diminish the accuracy of pointer analysis for stack and global pointers.

They implemented and tested their analysis over a suite of programs and present the results of their analysis. The largest program analyzed had 2279 lines of C code. Their results show that the average number of pointers arriving at an indirect memory operation is 1.13. This indicates that their analysis is very accurate, since each indirect memory operation must have at least one pointer reaching it. They also point out that indirect memory operations that can be determined to access only one location can be modified into explicit memory operations if the accessed location is visible to the function containing the operation. They also note that pointer analysis can be used for array dependence testing, instruction scheduling, parallelization and other optimizations. However, they do not actually show the impact of pointer analysis on subsequent optimizations.

3.4 Wilson and Lam

The goal of Wilson and Lam's work is to provide context-sensitive analysis efficiently [36]. Context-sensitivity takes time exponential in the program size in the worst case, but they make their implementation more efficient by exploiting the fact that the number of aliasing patterns that arrive at the input to a function is usually small. They exploit this fact by developing *partial transfer functions*. Transfer functions were initially developed to perform interprocedural constant propagation[18]. For constant propagation, a transfer function summarized a function's effect on constants. Constants arriving at a call site are passed to the callee's transfer function. The transfer function computes what constants will result from the given input. The benefits of using a transfer function are speed and context-sensitivity. Speed is gained because much of the analysis needed for constant propagation can be stored within the transfer function. This stored analysis can be applied to multiple call sites, thus gaining efficiency. Context-sensitivity is achieved since transfer functions are applied to each call site individually. Transfer functions are more difficult to create when applied to the problem of pointer analysis. Wilson and Lam give an example where the effect of a simple function depends heavily on the aliasing patterns that enter it. Thus, a transfer function that completely summarizes a function's effect on pointers may be as complicated as analyzing the function itself (*i.e.*, little analysis can be stored within the transfer function). Wilson and Lam construct efficient transfer functions for pointer analysis by limiting the initial aliasing patterns in which they are applicable, thus the name partial transfer function. When a call site to an unanalyzed function is encountered, a partial transfer function is developed for the function that is applicable to that context's aliasing pattern. Subsequent call sites to that function can re-use the partial transfer function if they satisfy the partial transfer function's initial aliasing pattern requirements. If these requirements are not satisfied, a new partial transfer function is created. Wilson and Lam's work depends on the fact that a single partial transfer function will frequently be enough to cover all the aliasing patterns that enter a function in a program. In their results for a given program, the average number of partial transfer functions created for a function ranges from one to 1.39.

By using partial transfer functions, they can safely use representative names (they call them "extended parameters"), instead of explicit names (see Section 2.2). Their work recognizes that aliasing may exist between representative names, and representatives cannot be created without regard to the aliasing patterns that may exist. By testing to make sure that the aliasing patterns at a call site satisfy the requirements of a partial transfer function, they do not have to create representatives assuming the worst case aliasing pattern. They handle the problem of creating representatives for arbitrary size data-structures by only creating locations as they are needed.

Wilson and Lam's answers to our other questions are similar to the ones we have chosen. They perform a points-to analysis rather than alias analysis, and they also choose to split the heap by call site. They note that splitting by call path can produce more accurate results, but it can create too many heap names. Wilson amd Lam tested their analyzer on a suite of programs that ranged in size from 188 to 4663 lines. For each program they present the time required for their analysis and the average number of partial transfer functions created for each function. Their results were used to parallelize two programs. They show the percentage of time that a sequential version of each program would spend in code that was parallelized. They also show the speedup achieved in the parallelized code.

3.5 Ruf

Ruf's work was to quantify the improvement in accuracy of pointer analysis due to using complete context information instead of no context information [32]. Ruf does not try to develop an efficient context-sensitive analysis. His goal is to determine the maximum possible benefit in order to provide a data-point for others who are considering adding context-sensitivity to their pointer analyzer. Most people have assumed that context information would greatly improve the accuracy of pointer analysis. Ruf's work directly compares two pointer analyzers that are identical except that one implementation uses complete context information. The context information that he chooses to use is *assumption-set-based contexts*. In this system he attaches a set of conditions with every points-to pair. These conditions are propagated with points-to pairs through a procedure. At loads and stores, if two pointers are involved, the conditions on both points-to pairs involved are combined to produce the output points-to pair. When a points-to pair is returned from a procedure to a call site, the points-to pair is only returned to call sites that satisfy its conditions. Ruf's work surprisingly shows that context information produces no increase in the accuracy of pointer analysis.

Ruf's choice for context information is different from the one we have selected for our work. Ruf chooses to associate conditions with each points-to pair while we associate a call-stack or path through the call-graph. He chooses this type of context information because it allows him to prune contexts based on the results of a context-insensitive analysis. He prunes contexts from points-to pairs at indirect memory operations that context-insensitive analysis has determined modify only one location, (*i.e.*, has only one points-to pair at the base register). In this case, all call sites must produce this points-to pair for the base register. Thus context information will not improve accuracy at all. Ruf also prunes away contexts at *strong updates* based on the results of context-insensitive analysis. In a context-insensitive setting, a strong update is a more accurate way of treating stores when it is known that the store *must* modify a certain location. Since the store must modify a specific location, any old pointer values that were in that location prior to the store can safely be killed (*i.e.*, removed). In a context-sensitive setting, strong updates can be partially achieved even if a store may operate on multiple locations. Old values at a store can still be killed, for those contexts in which a specific location must be modified. In this case, Ruf enumerates the contexts in which the pointer may survive and attaches it to the surviving pointer. Context-insensitive analysis can remove the need for enumerating contexts in the context-sensitive case for those locations which context-insensitive analysis has already shown will not be modified. It is not clear to us why these pruning techniques require the use of an assumption-set based approach to context information as opposed to a call-stack based approach.

One technique that we use in our context-sensitive analysis that Ruf does not use is to modify our propagation (when compared to the context-insensitive case) of points-to pairs at loads and stores based on the context information. In Ruf's work, when two points-to pairs arrive at a memory operation their conditions are combined to form the condition for the output pair. This may produce an unsatisfiable condition for the output (*e.g.*, the resulting condition may require one formal parameter to point to two different locations). With a call-stack based approach to contexts, this corresponds to the case where pointers with two different paths or call-stacks arrive at a memory operation. Since they must come from different contexts, they cannot interact in a real execution. Thus, in our analyzer we do not propagate the interaction of two pointers in such a case. This may improve the accuracy of our context-sensitive analysis over Ruf's work.

Ruf's work is similar to ours in many important respects. He uses a points-to framework for analysis as opposed to an alias framework. He also splits the heap by heap-allocating call sites. He does not use representative names (he calls them "synthetic" names) to handle call sites, but like us he uses explicit names. However, he does not explain the drawbacks of using representative names. He does note that using explicit names can reduce the opportunities for strong updates and thus reduce accuracy. On the other hand, he notes that explicit names can also improve accuracy by eliminating aliasing relationships unnecessarily generated by merging aliasing relationships from multiple call sites. Also, like our work, he gives performance results for his analyzer. The largest program he analyzes has 6771 lines. The analysis times for the context-insensitive version of his analyzer ranges from 1 to 35 seconds. The type of machine that the experiments were performed on is not specified. Also, he does not show what impact his analysis will have on later optimization passes.

3.6 Summary

Work on pointer analysis for C developed as an extension of the work done on alias analysis for FORTRAN programs. Early work in this area used alias pairs to analyze C programs. Later work has switched to using the points-to abstraction.

Early work in pointer analysis has not been very clear about how to model the behavior of pointers at call sites. Particular attention needs to be paid at these locations since pointers allow a function to modify its caller's memory. Wilson and Lam dealt with this by using representative names and partial transfer functions. Ruf dealt with this by using explicit names.

Previous work has also not been conclusive on what heap model to use. Various models have been proposed (*e.g.*, to use a single name, split by call site, split by call path, *k-limiting*, *etc.*). No numbers have been shown to quantify the tradeoffs among these models. Previous work has also not been conclusive about what form of context information to use, path expressions or conditions. Ruf's work suggests that context information is of no use in improving the accuracy of pointer analysis.

A summary of the properties of previous work is shown in Figure 3.1.

Pointer analysis by itself will not improve the performance of code. It is only beneficial if later optimizations can utilize the information it generates. All of the numbers presented in the previous work in this area (except Wilson and Lam's parallelization numbers) have been concerned with the performance and accuracy of the pointer analysis itself. No numbers showing pointer analysis' impact on the running time of analyzed programs have been published.

	Type	Call Sites	Heap	Path Information
LR	Alias	Representatives	*k-limiting* and	conditional
			call site	
CBC	Alias	Representatives	Call Path with *k-limiting*	conditional
EGH	Points-to	Representatives	One Name	arbitrary-level
WL	Points-to	Representatives	Call Path	arbitrary-level?
Ruf	Points-to	Explicit Names	Call Site	none/arbitrary-level
Lu	Points-to	Explicit Names	Call Site	none

LR	Landi & Ryder
CBC	Choi, Burke, & Carini
EGH	Emami, Ghiya, & Hendren
WL	Wilson & Lam

Figure 3.1 Comparison of approaches

Chapter 4

PL Style Analysis

In this chapter, we will describe the analyzer we implemented and examine its effectiveness. The analyzer uses pointers rather than alias pairs (see Section 2.1) and explicit names rather than representatives (see Section 2.2). The analyzer represents programs in an assembly-like language, and performs flow-sensitive analysis. Versions of the analyzer were made with various models for the the heap and different forms of path information (see Chapter 2 for a discussion of these choices). This chapter contains six Sections. Section 4.1 describes an abstract pointer language, *PL*. Section 4.2 shows how pointer analysis can be performed on *PL*. Section 4.3 explains how the pointer behavior of a C program can be modeled with a *PL* program. Section 4.4 describes our implementation of a *PL* style pointer analyzer for C. Section 4.5 discusses the sources of approximation in the analyzer. Section 4.6 summarizes our current work.

4.1 *PL* language

Here we define an abstract language *PL* that has the relevant features of pointer-based languages but is easier to analyze. We will use it as the abstract model upon which our pointer analyzer will be based. A *PL* program consists of:

- global regions G_i
- functions F_j, one of which, F_0, is the entry point into the program
- function regions $FR_{j,k}$, the k-th function region of function F_j

A function F_j creates function regions $FR_{j,k}$ when it is invoked. These regions are destroyed on exit. The syntax of *PL* is described in the following grammar:

function	$\rightarrow$ label **FUNCTION** (register*) [region*] block*
block	$\rightarrow$ label operation*
operation	$\rightarrow$ load $\mid$ store $\mid$ address $\mid$ call $\mid$ jump $\mid$ branch $\mid$ return
jump	$\rightarrow$ **JMP** label

branch	$\rightarrow$ **BR** label label
return	$\rightarrow$ **RTN** register [region*]
load	$\rightarrow$ **LD** register register [region*]
store	$\rightarrow$ **ST** register register [region*] [region*]
address	$\rightarrow$ **ADDR** [region | function] register
call	$\rightarrow$ [**CALL** label register register* [region*] [region*] |
	CALLR register register register*] [region*] [region*]
region	$\rightarrow G_i \mid FR_{j,k}$

Since *PL* is only concerned with pointer analysis, operations not relevant to pointer analysis have been left out. The branch operation is indeterminate, since we assume any intraprocedural path may be taken. Operations that may modify or reference regions (*i.e.*, calls, loads and stores) have been extended with lists that tell what regions the operation may reference or modify. For calls and stores, the referenced list comes before the modified list. If nothing is known about what regions may be accessed, these lists may contain every region. Our analysis will communicate its results by shortening these lists. In addition, a list of regions is also associated with the entry (**FRAME**) and return (**RTN**) of each function. These lists contain the set of regions that may be used in the function and set of regions that may be modified in the function respectively. The set of regions that may be used in a function is called its *Uses* set.

In a legal *PL* program, an instantiation of a function region $FR_{j,k}$ can only be referenced or modified by loads or stores if it is the topmost instantiation of $FR_{j,k}$. This is not a natural consequence of the syntax, but is a restriction the programmer must guarantee. This requirement is necessary in order to make pointer analysis simpler (see the explanation of how to propagate a region to a call site in Section 4.2). A consequence of this restriction is that ALVRs must be modeled by global regions (see Section 4.3).

4.2 Pointer analysis

Our goal will be to label all loads and stores in *PL* programs with the regions that they may reference or modify. A load or store labeled with a function region $FR_{j,k}$ indicates that the load or store may reference or modify the topmost instantiation of $FR_{j,k}$. We will accomplish this by converting our *PL* program into SSA form and then propagating pointer values through this SSA form.

4.2.1 SSA form

To label all the loads and stores, the program must be converted into SSA form[14]. This includes giving SSA names to all global regions and the topmost instantiation of all function regions. These regions, from the point of view of our SSA construction algorithm, are just like the registers that may be defined and used by an operation. All that is required by the SSA construction algorithm are the locations where a resource (region or register) may be referenced, where it may be modified, and where it is created. A region's reference and modification points are specified by the region lists associated with calls, loads, and stores. A function's *Uses* set serves as the creation point for the regions used in a function.

The SSA construction algorithm we use creates the SSA form for each function individually. It does not create an interprocedural SSA representation. Since we are performing an interprocedural analysis, we have a few additional requirements so that SSA names at the interfaces between functions can be easily connected. In an intraprocedural setting, a function's *Uses* set serves as a creation point for all the regions used in the function. In an interprocedural setting, the non-local regions in a function's *Uses* set are interpreted differently. For these regions, the *Uses* set serves as a join point. All the SSA names for a non-local region at the call sites to the function are joined at the SSA name created for the region at the function's *Uses* set. This serves the same purpose as joining SSA names from different basic blocks at a *phi*-node. The only differences are that we are joining SSA names from different call sites rather than different basic blocks, and we use the *Uses* set rather than a *phi*-node. If a non-local region is modified in a function, it acquires a new SSA name. These modifications are gathered together by placing all potentially modified regions in a referenced list at each return from a function. These new names are connected to the list of modified regions for each call site to the function. We should note that we only need to give SSA names to a region over its lifetime. Thus, the referenced and modified lists of calls, the referenced list of returns, and the *Uses* set of a function do not need to include regions that are created by descendants of the function that contains the list or set. These lists and the *Uses* sets should be as accurate as possible to reduce the time and space needed to build the SSA form of the program.

In summary, there are seven sets and lists that must be specified in order to generate the SSA form of a *PL* program. They are: referenced lists for loads, stores, returns and calls; modified lists for stores and calls; and the *Uses* set of a function. Figure 4.1 shows how these lists and sets might be specified for an example program. In this example, the *Uses* set for a function is just the union of all global regions and all local regions created by the function and its possible ancestors. All referenced and modified lists are just the *Uses* set for the function they occur in except for the referenced list of a return. The referenced list for a return in a function is just the *Uses* of the function minus all the locals created by the function. This is a conservative setting for the lists and the *Uses* sets. There are six important properties of this setting:

- all regions possibly referenced by a memory operation are in the memory operation's referenced list
- all regions possibly modified by a store are in the store's referenced and modified lists
- all regions used in a function are in the function's *Uses* set
- all regions possibly modified by a function are in the referenced lists of the function's return operations
- all regions possibly referenced by a call are in the call's referenced list
- all regions possibly modified by a call are in the call's modified list

See Section 4.3 to see how these lists and sets are further limited in our implementation.

4.2.2 Propagation

Every SSA name s will have a set of possible addresses it may contain written as $POINTS\_TO(s)$. The target of each **ADDR** operation is inserted into a worklist and initialized with the address of its region or function argument. The SSA name for each global region in the *Uses* set for F_0 is initialized with any addresses the region may have when the program starts. If the region has an address, it is placed on the worklist. We remove items from the worklist and propagate them to their uses. The algorithm terminates when the worklist is empty. To propagate addresses, the analyzer uses the function **SetCopy**(dst, src) to set $POINTS\_TO(dst) = POINTS\_TO(dst) \cup POINTS\_TO(src)$. If this changes $POINTS\_TO(dst)$, dst is added to the worklist.

Figures 4.2 and 4.3 illustrate the actions needed to propagate an address in a load operation. In this example, we are propagating to a load operation with base register `r11` and target register `r12`. The memory regions that may be accessed by this load are listed in brackets after the operation. They are: `@_ap`, `@_bp`, and `@_cp`. These memory regions have been given SSA names, which are denoted by the subscripts. The "address" of these memory locations is shown above the box that represents them. Thus, memory region `@_ap`'s address is 0. Propagation to this load begins

Globals: G_0

Function: F_0
Locals: $FR_{0,0}$

FUNCTION	$[FR_{0,0}, G_0]$	
LD	$[FR_{0,0}, G_0]$	
ST	$[FR_{0,0}, G_0]$	$[FR_{0,0}, G_0]$
CALL F_1	$[FR_{0,0}, G_0]$	$[FR_{0,0}, G_0]$
CALL F_2	$[FR_{0,0}, G_0]$	$[FR_{0,0}, G_0]$
RTN	$[G_0]$	

Function: F_1
Locals: $FR_{1,0}$

FUNCTION	$[FR_{0,0}, FR_{1,0}, G_0]$	
ST	$[FR_{0,0}, FR_{1,0}, G_0]$	$[FR_{0,0}, FR_{1,0}, G_0]$
CALL F_3	$[FR_{0,0}, FR_{1,0}, G_0]$	$[FR_{0,0}, FR_{1,0}, G_0]$
RTN	$[FR_{0,0}, G_0]$	

Function: F_2
Locals: $FR_{2,0}$

FUNCTION	$[FR_{0,0}, FR_{2,0}, G_0]$	
CALL F_3	$[FR_{0,0}, FR_{2,0}, G_0]$	$[FR_{0,0}, FR_{2,0}, G_0]$
RTN	$[FR_{0,0}, G_0]$	

Function: F_3
Locals: $FR_{3,0}$

FUNCTION	$[FR_{0,0}, FR_{1,0}, FR_{2,0}, FR_{3,0}, G_0]$
RTN	$[FR_{0,0}, FR_{1,0}, FR_{2,0}, G_0]$

Figure 4.1 Operations with lists of modified and referenced regions

by examining the contents of the base register, `r11`, of the load. We start with the element 0. We look for any memory regions in the list of regions that this load may reference that are located at address 0. We see that `@_ap`$_{13}$ represents `@_ap` at this point, and it is located at address 0. Since `@_ap`$_{13}$ represents a memory region that may be pointed to by the base register of this load, we copy the contents of `@_ap`$_{13}$, 113, and put it into the target register, `r12`. We follow an identical procedure with the other element, 4, in the base register and put the contents of `@_bp`$_{14}$, 97, into `r12`. Since the address of memory location `@_cp`$_{15}$ is not in `r11`, we do not copy its contents, 126, into `r12`. At the end of propagation the target register, `r12`, will contain 113 and 97. Since its contents have changed, `r12` will be added to the worklist.

The functions that perform propagation are shown in Figures 4.4 and 4.5. The steps needed to propagate addresses are fairly obvious except for (1) propagating a region to a store, (2) the treatment of regions at a call site, and (3) propagating a function pointer to a call site.

1. A region in the referenced list for a store must have its addresses added to the corresponding region in the modified list because a store might not kill values in a region (*i.e.*, some values might survive a store if the region references more than one possible name). Thus any addresses in a region before the store may still be in the region after the store. This is why stores need referenced lists.

2. There are two cases for handling a region r in the referenced list at a call site to function F_l. If r represents a region $FR_{l,n}$ that is created by F_l,[6] then r is copied directly to the modified list at the call site. Otherwise, r is copied into the corresponding region in the *Uses* set for F_l. In the first case, this is done because r represents the topmost instantiation of $FR_{l,n}$ before the call site. Within the call, it is no longer the topmost instantiation. Thus, it cannot be referenced or modified, and the values in the topmost instantiation of $FR_{l,n}$ before and after the call are the same. This is reflected by copying r from the referenced list to the modified list. In the second case, $FR_{l,n}$ is visible in the call, so its values must be copied into the call. If these values are modified by the call, the modified values will be placed in the call site's modified list when the called function's **RTN** operation is processed. A **RTN** operation's referenced list contains all non-local regions possibly modified by a call.

[6] If our MOD/REF analysis is powerful enough, this case will only occur if F_l is recursive. If F_l is not recursive then our MOD/REF analyzer should have been able to remove r from the call site's referenced list.

`LOAD r11 => r12` [@_ap$_{13}$ @_bp$_{14}$ @_cp$_{15}$]

	0	4	8
Memory Regions	@_ap	@_bp	@_cp

SSA Name	11	12	@_ap$_{13}$	@_bp$_{14}$	@_cp$_{15}$
Original Name			@_ap	@_bp	@_cp
Contents	0,4		113	97	126

Figure 4.2 Before propagation step

`LOAD r11 => r12` [@_ap$_{13}$ @_bp$_{14}$ @_cp$_{15}$]

	0	4	8
Memory Regions	@_ap	@_bp	@_cp

SSA Name	11	12	@_ap$_{13}$	@_bp$_{14}$	@_cp$_{15}$
Original Name			@_ap	@_bp	@_cp
Contents	0,4	113,97	113	97	126

Figure 4.3 After propagation step

3. When a function pointer is propagated to a call site, the calling-context must be propagated to the new target. In addition, the return context of the called function must be propagated to the call site. This may seem unnecessary, because the return context will be generated by the newly propagated calling-context. However, if the newly propagated calling-context does not modify the existing return context (the called function may have other call sites that may have already created a return context), the return context will not be properly returned.

4.2.3 Running time

Before analyzing the running time, it should be clear that this algorithm must halt. The algorithm continues only if an SSA name has a region added to its $POINTS\_TO$ set. Since there are a finite number of regions and SSA names, this process must terminate. Four parameters will be used to analyze the running time of the propagation algorithm.

- S Number of SSA names
- R Number of regions
- U Maximum number of uses of one SSA name
- D Maximum number of SSA names one operation can define

Each SSA name can be inserted and removed from the worklist at most R times. Thus, each SSA name will be processed at most R times. Each time an SSA name is processed, it may result in UD **SetCopy** calls. **SetCopy** can be implemented to require O(#source addresses) + O(#target addresses) time. Since both of the addends are of O(R), **SetCopy** requires O(R) time. The only non-constant work needed to determine if a **SetCopy** operation needs to be done is in the **LD** and **ST** cases. In these cases, there is a test, PointsTo(operation.BaseRegister,region). If the addresses in operation.BaseRegister are represented by a linked-list, this can be done in O(R) time. Since this overhead is of the same order as the **SetCopy** operation, the worst case running time is O(R^2SUD). The worst case requires that every SSA name contain every address upon termination and that each **SetCopy** call adds at most one address. The average case of our test suite is much better than the worst case (see Figure 4.6). Consider the results of our baseline analysis where the heap is split by `malloc` call sites and no path expressions are used. In the fourteen programs tested with our baseline analysis, there were a total of 2,822,907 SSA names when

```
Void PropagateToCall(Name ssaName,Operation callingOperation,Function calledFunction) {
    if (IsActualParameter(ssaName,callingOperation))
        SetCopy(CorrespondingFormal(ssaName,calledFunction),ssaName);
    else {
        /* propagating a region to a call */
        if (RegionIsLocalToFunction(ssaName,calledFunction))
            SetCopy(CorrespondingRegion(ssaName,callingOperation.ModifiedRegions),
                ssaName);
        else
            SetCopy(CorrespondingRegion(ssaName,calledFunction.UsesRegions),ssaName);
    }
}
```

Figure 4.4 Code for **PropagateToCall**

using a heap model where the heap was split by `malloc` call sites. Of these SSA names, 341,286, or 12% contained more than nine addresses. Over 79% of the SSA names had no addresses. Thus, our worst case time bound is much worse than the average case of our test suite. Usually, an SSA name will contain very few pointer values. The **indent** program was especially difficult to analyze because of a large array of structures that contained over 200 pointers. Since an array is approximated by one name in the analyzer, any access to this array would refer to all 200 pointers, greatly slowing down the analysis. The program **jpeg** was also difficult to analyze because of a large number of indirect function calls. Excluding **indent** and **jpeg**, only 38,702 SSA names out of 2,338,952 (or 1.65%) contained over nine addresses and over 88% percent had no addresses.

4.3 Modeling and analyzing C in *PL*

The pointer behavior of C programs can be modeled with the *PL* language. Global variables are modeled with global regions. Local variables that are not addressed or are not created by a recursive function can be modeled by a function region. These variables cannot be accessed unless they are the topmost instantiation of the variable.

```
Void Propagate(Name ssaName,Operation operation,Function f) {
    /* propagating ssaName to a use in operation in function f */
    switch (operation.type) {
        LD     :  forAllRegions(region,operation.ReferencedRegions)
                      if (PointsTo(operation.BaseRegister,region))
                          SetCopy(operation.definedRegister,region);
                  break;
        ST     :  if (ssaName==operation.BaseRegister | |
                     ssaName==operation.ValueRegister)
                      forAllRegions(region,operation.ModifiedRegions)
                          if (PointsTo(operation.BaseRegister,region))
                              SetCopy(region,operation.ValueRegister);
                  else /* propagating a region to a store */
                      SetCopy(CorrespondingRegion(ssaName,
                          operation.ModifiedRegions),ssaName);
                  break;
        RTN    :  if (ssaName==operation.returnRegister)
                      forAllCallerSites(callOperation,f)
                          SetCopy(callOperation.definedRegister,ssaName);
                  else {
                      /* propagating a region to a return */
                      forAllCallerSites(callOperation,f)
                          SetCopy(CorrespondingRegion(ssaName,
                              callOperation.ModifiedRegions),ssaName);
                  }
                  break;
        CALLR :   if (ssaName==operation.functionPointerRegister) {
                      forAllCallees(calledFunction,operation) {
                          forAllParameters(parameter,operation)
                              PropagateToCall(parameter,operation,calledFunction);
                          forAllRegions(region,operation.ReferencedRegions)
                              PropagateToCall(region,operation,calledFunction);
                          forAllOutGoingRegions(region,calledFunction)
                              SetCopy(CorrespondingRegion(region,
                                  operation.ModifiedRegions),region);
                      }
                  } else {
                      forAllCalledFunctions(calledFunction,operation)
                          PropagateToCall(ssaName,operation,calledFunction);
                  }
                  break;
        CALL   :  PropagateToCall(ssaName,operation,operation.calledFunction);
                  break;
    }
}
```

Figure 4.5 Code for **Propagate**

Program	Number of Regions	Number of SSA names	SSA names with X number of addresses			
			0-9	percent	0	percent
go	817	1305321	1305285	100.0	1256788	96.3
cachesim	262	65129	65129	100.0	56344	86.5
allroots	31	1044	1044	100.0	880	84.3
fft	84	5539	5539	100.0	4638	83.7
gzip	437	126063	126054	100.0	103822	82.4
clean	389	158197	158185	100.0	126739	80.1
water	274	11313	11313	100.0	9022	79.7
tsp	66	6788	6788	100.0	5214	76.8
bison	355	106624	106619	100.0	75158	70.5
dhrystone	46	2649	2649	100.0	1834	69.2
mlink	651	457240	456232	99.8	384653	84.1
bc	282	93045	55413	59.6	48274	51.9
indent	312	166640	70697	42.4	52615	31.6
jpeg	378	317315	110674	34.9	106397	33.5

Figure 4.6 *POINTS_TO* set sizes

On the other hand, ALVRs[7] must be modeled by a global region. Since they are recursive, multiple instantiations may exist, and since they are addressed, we have a way to access instantiations that are underneath the topmost one. Thus, we cannot guarantee that only the topmost instantiation of these variables will be referenced. An SSA name for a function region only contains what the topmost instantiation of that region may have at that point, which is not sufficient to model ALVRs. An SSA name for a global region representing an ALVR contains the possible addresses of any instantiation of that variable at that point in the program. Since it represents multiple locations, addresses in these regions can never be killed. Fortunately, ALVRs in C appear to be rare. Heap memory is also represented by global regions. Stores to heap

[7] Of the programs tested, only **mlink** had an ALVR. ALVRs are easy to detect. A call-graph is built and all recursive functions are found. All addressed memory regions in these functions are ALVRs. It is complicated to model ALVRs more accurately. Accurate modeling would produce little benefit, since ALVRs are rare. Also, any attempt to model them more precisely would require complex code to be written that would slow down the performance of the analyzer even if no ALVRs existed in the program. Separate addresses would have to be created for the multiple regions used to model an ALVR. Complicated code would be needed to adjust these addresses when they enter and leave the function in which they are created.

memory cannot kill old values. In Figure 4.7, we have a short program that has one ALVR. There are four variables in this program: `p`, `alvr`, `not_alvr`, and `a`. Variable `a` is addressed but is not an ALVR since there can only be one instantiation of it. All accesses to it must refer to this single topmost instantiation. The variables `p` and `not_alvr` may have multiple instantiations, but only the topmost instantiation may be accessed, since these variables are unaddressed. Variable `alvr` may have multiple instantiations and is also addressed. Thus, it is an ALVR. In Figure 4.8, we have a program with seven regions. Each non-structure field of a structure is considered a variable and given its own region name. If a field of a structure is itself a structure, (*e.g.*, `g.d` in function `main`) then the naming is based on the fields of the interior structure. Note that variable `a` in function `f` is an example of an ALVR and is given a global region.

Calls to heap generating functions can be modeled with an **ADDR** operation. This splits the heap into sections based on the call site.

PL could be extended to handle explicit stores better. Stores in *PL* require a referenced list in order to propagate old values that may not be killed (see the discussion in Section 4.2.2 on the steps needed to propagate a region at a store). Since an explicit store is unambiguous about what region is being modified, we know that old values in the region will be killed. Thus, there is no need to build a referenced list for explicit stores. *PL* was not extended in this way in order to keep it simpler.

PL is too simple to model the generation of addresses in C. The intricacies of address creation are abstracted away in *PL* with an **ADDR** operation. For certain references (*e.g.*, explicit references) address generation in C can be adequately modeled with an **ADDR** operation. In these cases we know exactly which address is being created, and thus we know what region argument to place in the **ADDR** operation. On the other hand, an **ADDR** operation is inadequate for modeling the creation of addresses by arithmetic on a pointer value. Consider the example in Figure 4.9. Since we do not know whether `p` points to `a` or `b`, we do not know whether to generate an **ADDR** operation to `a.val` or to `b.val`.

We can model the generation of addresses more accurately in *PL* by assigning each region a numerical address. These addresses can be selected so that the addressing arithmetic in the program can generate addresses correctly. For example, suppose in Figure 4.9 that structures `a` and `b` are located at address 100 and 200, respectively, and that the `val` field has an offset of 8 in these structures. Thus, to generate this address we will have an operation like:

```
iADDI  8 r10 => r11
```

```
void recursive(int *p) {
  int alvr,not_alvr=*p;

  alvr=not_alvr+1;

  if (not_alvr==5)
    recursive(&alvr);
}

main() {
  int a=0;

  recursive(&a);
}
```

Figure 4.7 Program with one ALVR

Register `r10` represents the pointer variable `p`. Thus, `r10` will contain the addresses 100 and 200. Using the address calculation `r11` will contain the addresses 108 and 208, the addresses for `a.val` and `b.val`.

In order to use the program's addressing arithmetic to generate addresses for our regions, we assume that our front-end calculates addresses with a certain code shape. We require that address calculations be rooted at a stack pointer, global label, or address returned from a heap-allocating library call and that they consist of two separate components. The first component takes in the root address (*i.e.*, stack pointer, global label or address returned from a heap-allocating library call) and uses it to calculate the base address of the region being accessed. We require that each intermediate address calculated in this first component must also be an address to the base of a region. The second component is non-empty only for accesses to heap or array memory. It is used to access the interior of such regions. An example showing two ways to access the same memory location, one ill-formed and one well-formed, is shown in Figure 4.10. In this example we have two memory regions, `@_g` and `@_f`, both of size 8. Memory regions `@_g` and `@_f` appear in sequence. The ill-formed code creates a pointer to `@_g` and then increments it to access a location within `@_f`. This code merges the tail of component one with component two. The operation

```
struct inner {
  int a;
  int b;
};

struct outer {
 struct inner c[2];
 struct inner d;
};

main(int argc, char **argv) {
  int *a;
  struct outer g;

  a=(int *) malloc(sizeof(int));
  f(0);
}

void f(int a) {
  int *p;

  p=&a;
  if (a<5)
    f(a+1);
}
```

`main`'s function regions	@_a	@_g.c	@_g.d.a
	@_g.d.b		
`f`'s function regions	@_f.p		
global regions	@_heap	@_f.a	

Figure 4.8 Program with seven regions

```
if (a_or_b) {
  p=&a;
} else {
  p=&b;
}

f=p->val;
```

Figure 4.9 **ADDR** operation is inadequate

`iADDI 12 r1 => r2` needs to be split into two operations to divide the calculation into the two components. This is done in the well-formed calculation. Our front-end originally generated ill-formed code for accesses to a stack array with an index that was a compile-time non-zero constant.

We require address calculations to follow this format because we must be careful in how we handle addressing arithmetic for arrays. Unless we limit the addresses in some way, code that increments a pointer through an array will generate an infinite number of addresses. Our solution is to limit legal addresses to those that point to the base of a region. If a memory operation may access a location within a region, then the address to the base of the accessed region should be in the base register of the memory operation. In order to achieve this we must modify our address calculation algorithm from that of the real program in two ways. First, if an operation produces an address that is not the base of a region, then the computed address is discarded, and the input address is returned. The reasoning for this can be divided into two cases. First, this input address could not be a valid input address if this operation were part of the first component of an address calculation, since operations in the first component must produce addresses that are at the base of a region. In this case, no address needs to be returned. If this operation is part of the second component of an address calculation, then the input address should be returned. Since we cannot distinguish between these two cases, the input address is returned. Our second modification is to change the action of operations on addresses that point to heap memory and arrays. We identify addresses of all memory regions that are aggregates. Arithmetic operations on these addresses return the input address and may also return the computed address. The reasoning behind returning the input address is that the addressing arithmetic may compute a reference to a location within the array or heap. This is done even if the

```
iGLOBAL 0 @_g _g
_g: iDATA 0 1

bGLOBAL 0 @_f _f
_f: bDATA 0 8
```

Unacceptable Address Calculation	**Acceptable Address Calculation**
`iLDI    _g => r1`	`iLDI    _g => r1`
`iADDI   12 r1 => r2`	`iADDI   8 r1 => r2`
	`iADDI   4 r2 => r3`
`bPLDor  @!  1 0 r2 => r3`	`bPLDor  @!  1 0 r3 => r4`

Figure 4.10 Good and bad addressing calculation

computed address lands at the base of another region. This is necessary because array arithmetic may create pointers outside of the range of the array, (*e.g.*, if we create a pointer to the end of the array in order to access the array with negative indices). The computed address is returned if the computed address is at the base of a region. The reason for returning the computed address is that the computation may be part of the first component of an address calculation.

Thus, to generate addresses, we must add a great deal to *PL*, including numerical addresses, array addresses, addressing arithmetic, and region sizes. We did not do this because it is orthogonal to the propagation issues we wanted to examine with *PL*. It is also a very complicated extension. Generating addresses with a numerical framework was followed in our actual implementation.

Our rules for address calculation can be violated by some C code. Code that casts pointers to structured local or global memory may not follow our requirements (see Section 8.4 for an example of this).

4.4 Implementation

We implemented a *PL* style analyzer within the Massively Scalar Compiler Project (MSCP)[3]. We will describe it in five sections. The first four: C to `ILOC`, MOD/REF, propagation, and annotation, describe pointer analysis with no path expressions. The last section describes the modifications needed to perform analysis with path expressions.

4.4.1 C to `ILOC`

C2I translates C code into our intermediate language `ILOC` (see Appendix A). C2I generates a region for each scalar and array variable. Each field of a structure and each compiler-generated location is also given a region. C2I also records if a region is a global region or a function region, if a region is ever addressed, and the region's size. It also creates a region for each call site that returns heap memory. This memory is marked as both global and addressed. C2I can explicitly annotate a load or store that does not use a pointer with the region that it will reference or modify. C2I is also able to denote that some stores must kill old values in the region the store modifies. Figure 4.11 shows a C source code fragment and the `ILOC` that C2I produces for it. This example contains four regions. Global regions are created for `g1` and each field of `g2`. A local region is created for local variable `c` in function `f`. ALVRs are created as local regions by C2I since C2I will not have enough information to determine that an addressed local is an ALVR in the general case (*i.e.*, the fact that a local is an ALVR may depend on multiple files). Later when our analyzer processes all the files, it can determine if a local is an ALVR. In this case, our analyzer will process the ALVR as a global region, even though it was declared with a `STACK` statement. Locals, including ALVRs, are declared with a `STACK` declaration. Ambiguous memory operations are tagged with '@!'. Later, our analyzer will create group tags to more accurately tag ambiguous memory operations. The other three tags `@_addr_locals`, `@_writable_globals`, and `@_addr_globals` provide information used by our analyzer to perform MOD/REF analysis. They are not used to tag operations.

4.4.2 MOD/REF analysis

MOD/REF analysis is crucial to the success of pointer analysis. It is necessary to limit the number of SSA names generated for memory regions. Since many of our test cases are limited by the amount of memory used by our analyzer, we have strived to make our MOD/REF analysis as accurate as possible in order to trim our analyzer's memory requirements. See Figure 5.3 for the time and memory needed to produce the SSA form for our test cases without MOD/REF analysis.

```c
int g1;
n
struct pair {
  int a,b;
} g2;

void f() {
  int *p,c,d;

  p=&g1;
  *p=3;
  p=&c;
  d=3;
  g2.b=*p;
}
```

```
_f:     0       FRAME   8 => r0 r1 [ i i ]
        10      iLDI    _g1 => r4
        10      i2i     r4 => r2
        11      i2i     r2 => r5
        11      iLDI    3 => r6
        11      iPSTor  @! 4 0 r5 r6
        12      iADDI   0 r0 => r8
        12      i2i     r8 => r2
        13      iLDI    3 => r9
        13      i2i     r9 => r3
        14      iLDI    _g2 => r10
        14      iADDI   4 r10 => r10
        14      i2i     r2 => r11
        14      iPLDor  @! 4 0 r11 => r12
        14      iSSTor  @_g2_4 4 0 r10 r12
        15      RTN     r0
        ALIAS @_addr_locals [ @f_c_0 ]
        iSSTACK 0       @f_c_0 _f
_g2:    bDATA 0 8
_g1:    bDATA 0 4
        ALIAS @_writable_globals [ @_g1_0 ]
        ALIAS @_addr_globals [ @_g1_0 ]
        iSGLOBAL        0       @_g1_0  _g1
        ALIAS @_writable_globals [ @_g2_0 ]
        iSGLOBAL        0       @_g2_0  _g2
        ALIAS @_writable_globals [ @_g2_4 ]
        iSGLOBAL        4       @_g2_4  _g2
```

Figure 4.11 C code translated to ILOC

The first step in MOD/REF analysis is to rewrite the modified and referenced lists for each ambiguous memory operation. Four restrictions are used to limit the memory regions that may appear in these lists. First, non-integer ambiguous memory operations have empty lists. Although these operations reference memory, we assume they cannot move a pointer value into or out of memory. Since we are only tracking pointer values, we do not create new SSA names for memory operations that cannot affect the state of our analysis. Although we do not create SSA names for the memory regions these operations may access, this does not mean that our pointer analyzer cannot later determine that these operations will access some memory regions. Second, only addressed locals and globals may be affected by ambiguous loads and stores. This information is provided by C2I. Third, non-writable globals cannot be modified by a store. An example of a non-writable global is a string literal passed to a `printf` call. Fourth, a local can only appear in memory operations that are in descendants of the function in which the local is created.

Once we have limited the MOD/REF effect of ambiguous memory operations, we calculate the MOD/REF effect for whole functions. The MOD/REF effect of a function is composed of two parts:

- the local MOD/REF effect
- the MOD/REF effect of the functions it calls

The local MOD/REF effect is determined by unioning together the MOD/REF effects of all the memory operations in the function. For ambiguous memory operations, their MOD/REF effect is limited by the rules in the previous paragraph. The second component of a function's MOD/REF effect is determined by calculating the strongly connected components (SCC) of the call-graph. The call-graph is formed by assuming that an indirect function call may call any addressed function. The MOD/REF effect for every function in an SCC is the same since the functions are mutual descendants. The SCCs are processed in reverse topological order so that the MOD/REF effect of any called function not in the SCC being processed has already been calculated.

In writing the referenced and modified lists of procedure calls and returns, we require that the memory regions in a call site's referenced list have the same size and order as the non-local memory regions in the *Uses* set (see Section 4.2) for the called function. Similarly, we require that the list of referenced memory regions for a return match the modified list for the corresponding call sites. These requirements are necessary to allow efficient pointer analysis. When transferring information between call sites, we do not want to search for a corresponding name in a *Uses* set or a modified list. We accomplish this by ensuring that the same offset used to access a memory region in a call or return's referenced list can be used to access the new SSA name for a memory region in the *Uses* set or modified list. This requirement forces us to use the same MOD/REF set for all call sites to a function. In particular all explicit calls to an addressed function, say `f`, and all indirect calls (prior to pointer analysis, we assume that an indirect call may call any addressed function, including `f`) must have the same MOD/REF effect. Since MOD/REF analysis assumes that an indirect call may call any addressed function, the MOD/REF effect for any indirect call site is the union of the MOD/REF of all addressed functions. Thus, the MOD/REF effect for explicit calls to `f` must also be the union of the MOD/REF effect of all addressed functions. This lowers the accuracy of our MOD/REF analysis for explicit calls to an addressed function. Our requirements for matching also force all addressed functions to have identical *Uses* sets and referenced lists for their returns. Note that these size and ordering requirements are not necessary when we redo MOD/REF analysis after pointer analysis. Thus, we are free to tailor the MOD/REF effect for each call site to a function by using path information (see Figure 2.10).

MOD/REF analysis not only reduces the memory requirements of pointer analysis, it can also improve the accuracy of pointer analysis. In Figure 4.12, `p` will contain the address of `g` before the call to `printf` in `f1`. Without MOD/REF analysis, this fact will enter the `printf` function, because all variables are in `printf`'s referenced and modified lists. The variable `p` will be propagated to the return operation in `printf` and will be returned to all call sites of `printf`. (Note that this is an example of the false-return problem.) Thus, `p` will unnecessarily point to `g` immediately after the call to `printf` in `main`. With MOD/REF analysis, we know that `printf` cannot modify or reference `p`, so it will not appear in `printf`'s referenced and modified lists. Thus, `p` will not be propagated into `printf`, nor will it be propagated back to `printf`'s call sites.

4.4.3 Propagation

The actual pointer analysis starts by assigning a numerical address to each region. These addresses are chosen so that an add immediate operation generated to access a local variable in a stack frame can generate the correct address by adding its immediate operand to the address in the base register. The addresses of regions in a structure are also chosen so that the addressing arithmetic is preserved. Care is taken in handling add immediates to ensure that only addresses to the start of a region are generated. Otherwise, an infinite number of addresses might be generated by code that increments a pointer through an array. Using numerical addresses also makes initialization of the worklist easier. All references to a stack frame can be initialized by inserting the base register onto the worklist. Once the SSA names are initialized

```
int *p,g;

void f1() {
  p=&g;
  printf("hi");
}

main() {
  printf("hello");
  f1();
}
```

Figure 4.12 MOD/REF analysis improves accuracy

with these addresses, propagation is started. Propagation is done as described in Section 4.1. In our implementation, the set of pointer values associated with an SSA name (its $POINTS\_TO$ set) is kept in a linked-list. When no path expressions are used, the pointer values are moved to a vector set if the size crosses a threshold of ten (when path expressions are used, only the linked-list representation is used) [6]. The reason for using multiple representations is to reduce the memory requirements of the analysis. Figure 5.4 shows the time and memory requirements if an exclusively linked-list representation is used. These results were obtained by setting the threshold to 1000. Since this was larger than the maximum $POINTS\_TO$ set size for any of our programs, it produced the desired results.

4.4.4 Annotation

Once propagation is finished, ambiguous loads and stores are re-annotated with more precise lists of the memory regions that they may use. This is done by looking at the addresses in the base registers of such operations and finding the regions at those addresses. MOD/REF analysis is performed again to re-annotate the referenced and modified lists for calls and return operations. There are four differences between the MOD/REF analysis performed before propagation and the one done after propagation. First, ambiguous loads and stores have had their MOD/REF effect limited by pointer analysis. Second, the targets of indirect function calls have been limited by pointer analysis. Third, we do not require that a call site's referenced list match the *Uses* set of its target, nor do we require that a return's referenced list match a corresponding call site's modified list. This requirement was only necessary to speed up propagation. Since pointer analysis is finished, we are free from this restriction. Fourth, path information is available to tailor the MOD/REF information for a function to a particular call site (see Figure 2.10) or call path.

4.4.5 Other details

In order to perform pointer analysis, our analyzer expects an `ILOC` definition for all functions used in the program. In particular, it needs a definition for all library functions called by the program. These definitions cannot be created by C2I since we do not have the C source files for many of these functions. Our solution is to write hand-made `ILOC` files that have the same MOD/REF and pointer effect as the library functions. These hand-made files do not have the same functionality. Figure 4.13 shows our hand-made `ILOC` file for the `atoi` function. The `atoi` function takes in a pointer to a string and returns the decimal value of the string as an `int`. From a MOD/REF perspective, `atoi`'s only effect is to reference its string argument. We simulate this effect by loading the string with the `iPLDor` operation. We also return a `NULL` value in case the return value is used as a pointer.

4.4.6 Path expressions

We also implemented a version of our pointer analyzer that used path expressions. Path expressions were implemented by associating a list of call sites with each pointer value. The implementation allowed the user to specify the length of these expressions. To associate a path expression with each pointer value, we represented our sets of pointer values with linked-lists. Each node in the linked list contained a pointer value and a path expression. A vector set representation of our pointer sets was not possible since this did not allow path expressions to be easily associated with a pointer value.

```
_atoi:          0       FRAME   0 => r0 r1 r2 [ i i i ]
                0       iPLDor  @! 4 0 r2 => r9
                0       iLDI    0 => r20
                3       iRTN    r0 r20
```

Figure 4.13 Hand-made `ILOC` for an `atoi` function

Path expressions are maintained at call sites and returns of functions with multiple call sites. If only one call site exists for a function, maintaining path information for that call site would provide no help in tracking pointer values. By eliminating path information for these call sites we effectively increased the path-length of our analysis without increasing the space requirements. When a pointer and path-expression pair pass through a call site to a function with multiple call sites, the path expression must be modified. The new call site is put on the top of the new path expression, and the bottom call site of the path expression is popped off. At returns, the reverse process is done. The top call site is popped off. However, we do not know what call site to put at the bottom of the path expression, so a special marker representing any call site is put on the bottom. An example of maintaining path expressions is shown in Figure 4.14. In this example, path expressions are enclosed in angle brackets. The most recent call site in each path expression is on the right side. Each path expression is associated with a pointer value, and these pairs are enclosed in parenthesis. On the top of this example, we have a path expression, <`a`,`b`,`c`>, that reaches a call site `d`. This produces the new path expression, <`b`,`c`,`d`>. Note that `d` was added to the right since it is the most recent call site. Since the old path expression was already full, `a` was removed to create space for `d` in the new path expression. The bottom example shows the effect of returning this path expression. This path expression can only be returned through call site `d`. At the return, we remove call site `d`. Since we have no information about what call site preceded call site `b`, we put ANY in the leftmost position to signify any call site.

We use path expressions to modify our propagation algorithm in two ways. First, a pointer value is only returned to the call site that is at the top of its path expression. This will eliminate the false-return problem for those pointer values that do not have ANY as their top call site (see Figure 2.8). If the top call site is ANY then propagation proceeds just as if no path expressions are used.

Path expressions also affect propagation at loads and stores. The path expressions of the two pointer values involved (*i.e.*, base register and pointer value to be

$$(\mathrm{pt},\langle \mathtt{a},\mathtt{b},\mathtt{c}\rangle) \xrightarrow{\text{call site } \mathtt{d}} (\mathrm{pt},\langle \mathtt{b},\mathtt{c},\mathtt{d}\rangle)$$

$$(\mathrm{pt},\langle \mathrm{ANY},\mathtt{b},\mathtt{c}\rangle) \xleftarrow{\text{return to } \mathtt{d}} (\mathrm{pt},\langle \mathtt{b},\mathtt{c},\mathtt{d}\rangle)$$

Figure 4.14 Maintaining path expressions moved to or from memory) in the memory operation must intersect for propagation to occur. We test for intersection by checking that corresponding call sites in the path expressions are equal or that one of the call sites is ANY. This additional test can eliminate the pointer-mixing problem (see Figure 2.9).

Path expressions also modify the way that MOD/REF analysis is done after pointer analysis. Without path expressions the MOD/REF effect of a memory operation is propagated to all the callers of the function containing it. Path expressions can limit this MOD/REF effect to subsets of all the callers (see Figure 2.10).

4.5 Sources of approximation

There are seven sources of approximation in the analyzer:

1. Interprocedural Paths: The first source of approximation occurs because we have limited or no knowledge of the interprocedural path through which a pointer came. This lack of information causes inaccuracies when pointers are propagated to return operations. With limited or no path information, pointers may be passed in through one call site and returned to another call site. This is the false-return problem (see Figure 2.8). Lack of interprocedural path information can also create inaccuracies when we propagate to memory operations. Two pointers from different interprocedural paths cannot interact in a memory operation in a real execution of a program. Unfortunately, without complete path information, our analysis may allow two pointer values from different call sites to interact. This is the pointer-mixing problem (see Figure 2.9).

2. Intraprocedural Paths: The second source of approximation occurs because we do not record the intraprocedural path through which a pointer comes. This source of approximation causes inaccuracies when we propagate to memory operations. This inaccuracy is the intraprocedural analog of the pointer-mixing problem. Since the flow of control in a function is not stack-like, there is no intraprocedural analog of the false-return problem.

3. Stores: The third source of approximation occurs because multiple pointer values may reside in a single SSA name. This affects store operations since we cannot kill old values unless we know that a store modifies a specific scalar memory location. For pointer-based stores, we never kill old values, even if there is only one pointer to a scalar memory location in the base register. This was due to the difficulty in implementing it efficiently and the ineffectiveness of this technique when tested with an inefficiently implemented version. With interprocedural and intraprocedural path information, it is possible to kill old

values even if multiple pointer values reside in the base register. The key is to look for a pointer whose path does not intersect the path of any other pointer in the base register. If such a pointer p with path y is found, then we can optimistically assume that all execution paths described by y must store to p's target m. All pointers in memory location m whose path is contained in y can safely be killed. Later, if propagation places another pointer in the base register that intersects path y, then pointer values in m will have to be resurrected.

4. Heap: The fourth source of approximation is in the representation of the heap. We tried three models for the heap in our experiments: a single name, split by `malloc` call sites, and split by user heap allocator. All three of these models use a finite number of names to approximate an arbitrary number of locations. Obviously, some names may have to represent multiple heap locations. Thus, heaps and arrays are similar in that a single name may have to represent multiple locations. Heaps and arrays also cause inaccuracies in the same way. Loads from a heap name must load from all the locations represented by that name. Stores to a heap name cannot kill old values.

5. Arrays: The fifth source of approximation is in the representation of arrays. Since we do not perform any array section analysis, this will cause inaccuracies when we propagate to memory operations. Loads will load all the pointer values in an array. This is the main source of approximation when analyzing the program **indent** in our test suite. Also, pointer values in a store can never be killed.

6. Pointer Arithmetic: The sixth source of approximation is pointer arithmetic. Two values may be generated by an operation performing pointer arithmetic for a single input value. If the input pointer value pointed to an array or heap, then the input pointer value is returned. The reason for this is that the operation may just be accessing an array element within the array or a location within the heap name. In this case, returning the input pointer value is the correct action. If the computed value is the base address of a new memory location, then the computed address is returned. This handles the cases where a field of a non-heap structure or a stack allocated variable is being accessed. Both of these cases cannot occur simultaneously in a real program, but our analysis cannot determine which case should occur in some circumstances. Thus, it will return both addresses in order to be safe, but this will introduce inaccuracies. One circumstance in which this can occur is when pointer arithmetic is done on a heap pointer. We have arbitrarily set our heap size to 4000. However,

some arithmetic may generate a value that accidentally hits another memory location. A heap size of 4000 should make this unlikely, but it is still possible.

7. ALVRs: The seventh source of approximation is caused by ALVRs. Names for ALVRs must represent all instantiations. This is the only approximation that recursion forces in the analysis. Recursion by itself will not cause any approximation in the analysis. The approximation is needed for code like the code shown in Figure 4.15. Suppose function `f` was called with `f(0,NULL)`. This will result in recursive calls to `f` to a depth of three. All three calls will have a local instantiation of `a`. The call to `f` at depth zero has its instantiation of `a` addressed and passed to the other calls. The call at depth one stores through its instantiation of `a` to `global1`. The call at depth two sets the instantiation of `a` at level zero to point to `global2`. The analysis will unnecessarily conclude that the store through `a` at depth one at line `6` may modify `global2`. This occurs because `p` initially points to `a` due to the call on line `3`. Thus, line `8` will set `a` to point to `global2`. Upon exit from `f`, `a` will point to `global1` and `global2`. We unnecessarily conclude on line `6` that `a` may point to `global2`. This error occurs because all instantiations of `a` are represented by one name. It may be possible to get more accurate analysis by allowing multiple instantiations of a name (*e.g.*, `a_level0`, `a_level1`, `a_level2`, `a_level3`) in our analysis. However, this analysis would be very difficult. It would require knowing that certain portions of code only operate at a certain recursion depth. Furthermore, we have not seen code where this would be useful in our test programs.

These seven sources of approximation may generate other inaccuracies. For example, inaccuracies in the function pointer of an indirect function call will unnecessarily cause the calling context of the call to be merged with the initial context of the targets of the inaccurate function pointers. We do not consider these compounding effects to be a *source* of approximation. Of the seven sources of inaccuracies, the first five (interprocedural paths, intraprocedural paths, stores, arrays, and heap) are the most important. ALVRs rarely occur in real programs. In our test suite only **mlink** had one. Addressing arithmetic should rarely generate inaccuracies. In our implementation, a heap pointer can be incremented to hit the base of another region if it is incremented by a value of greater than 4000. This should be very unlikely. An array pointer can only hit the base of another region if the real program increments an array pointer to access a location outside of the array, or if the array pointer reaches a location that it could never reach in a real execution. Of the first five sources of approximation, arrays are probably the least important, although for certain programs, like **indent**, this source is crucial. Without the complicity of other sources

```
int global1,global2;

int f(int depth,int **p) {
  int *a;

  a=&global1;
  if (depth==0) {
    f(++depth,&a);
  } else if (depth==1) {
    f(++depth,p);
    *a=3;
  } else if (depth==2) {
    *p=&global2;
  }
}
```

Figure 4.15 ALVR causes approximation

of inaccuracies, arrays can only cause inaccuracies if the real program stores pointers in an array. Our feeling is that, because this construction is rare, the other four sources are more important. The order of importance of the remaining four sources is not clear. Techniques have been examined in our work to reduce the impact of all four of these sources except for intraprocedural paths. The technique for handling inaccuracies due to interprocedural paths, path expressions, is the most satisfying in terms of effectiveness. It can potentially completely handle all of the inaccuracies due to this source, if the program is non-recursive. However, path expressions can greatly increase the running time of analysis. Partial techniques have been developed to deal with inaccuracies due to stores and the heap. We can kill values with a store only in the clearest cases. A great deal of work has been done to refine the heap model[15, 10, 25, 20]. In fact, most of the work on pointer analysis is focussed on this source of inaccuracy. We have not seen any technique that deals with the inaccuracies due to intraprocedural paths.

4.6 Summary

In this chapter, we have developed an abstract language *PL* which can model the propagation of pointers in C. It does not model the address arithmetic used to create pointers in C. Using this model, we developed a pointer analyzer for C. This analyzer requires MOD/REF analysis in order to generate the SSA form of the program. It uses the points-to relation as its basic unit of information and explicit names to handle call sites. It also supports a variety of heap models and path expressions. We also showed that the context-insensitive version of our analysis has a polynomial time bound for its propagation step. We also identified seven sources of error in our analyzer.

Chapter 5

Results

We gathered experimental results on the analyzer and the analyzed code it produced by running test cases on a SPARC 10 with 128M of memory. The analyzer was tested on 14 programs (see Figure 5.1). The Lines column is the number of lines in all the source files used to make the program. This does not include header files. The number of lines for **clean** includes library files that contain functions not used in **clean**. We performed three categories of experiments with each program:

1. Analyzer performance - We measured the analyzer's performance under a variety of conditions:
   - with and without MOD/REF analysis
   - with and without multiple representations for points-to sets (*i.e.*, linked-lists and vector sets)
   - with various amounts of context information: path expressions ranging from length zero to four
   - with three heap models: split by user heap allocator call sites (LARGE), split by call site (MEDIUM), single name (SMALL)
2. Analyzed code performance - We measured the performance of compiled programs as a function of the analyzer's configuration. Configurations included:
   - no analysis
   - MOD/REF analysis
   - pointer analysis
     - with various amounts of context information: path expressions ranging from length zero to four
     - with three heap models: single name (SMALL), split by call site (MEDIUM), split by user heap allocator call sites (LARGE)

We also obtained cache statistics for the performance of our programs in our test suite. Statistics were obtained for programs with no analysis, MOD/REF analysis, and pointer analysis with no path expressions and a baseline heap model.

3. Optimizer performance - We measured the impact of different configurations of the analyzer on the performance of the rest of the optimizer.

5.1 Analyzer performance

In our tests of our pointer analyzer we attempted all feasible versions of our pointer analyzer for each test case. We allowed the length of path expressions to range from zero to four. We also allowed three models for the heap. In the SMALL heap model, a single name was used to represent the whole heap. In the MEDIUM heap model, a separate name was given for each call site to a library function that allocates heap space. In the LARGE heap model, a separate name was given for each call site to a user-defined heap allocator and for each call site to a heap-allocating library function. Note that each call site to a heap-allocating function only returned one value, the heap name for that call site. Even if another value reached the return-site of these functions, the value was not returned. This prevented heap names that were recycled from being propagated to every allocation call site. For some programs it was not possible to test with all varieties of the analyzer. This was due to two reasons. First, more precise models for the heap and longer path expressions both required more time and space in order to complete the analysis. Second, only five programs had user-defined heap allocators. A summary of the versions of our pointer analyzer that were used to analyze each program is shown in Figure 5.2. The numbers show the longest path expression that could be used to analyze the specified program and heap model. An "x" indicates that no experiments were performed in that category.

5.1.1 MOD/REF's impact on analysis

Our first test of the analyzer's performance was to determine the impact of MOD/REF analysis on the time and space requirements for the analyzer. Figure 5.3 shows the results of this experiment. In this experiment we only generated the SSA form of the program. We did not do the complete analysis. The MEDIUM heap model was used in this experiment. The "Memory" column shows the maximum amount of memory used to generate the SSA form without MOD/REF analysis. The "Time" column shows the amount of time necessary to build the SSA form without MOD/REF analysis. The "SSA" column shows the number of SSA names in the SSA form of the

Program	*Lines*	*Description*	*Inputs*
tsp	760	a traveling salesman problem	nine city tour
mlink	9264	genetic linkage analysis	cleft lip data set
fft	1037	fast-fourier transform	16x16x16 array
clean	11191	basic-block cleaning pass from MSCP	116 line `ILOC` function
cachesim	2849	cache simulator	161K instruction trace
dhrystone	534	classic benchmark code	2000 iterations
water	1345	from SPLASH benchmark	64 molecules, 2 time steps
indent	5955	prettyprinter for C programs	264 line input
allroots	215	polynomial root-finder	cubic polynomial
bc	7583	calculator language from GNU	program to find primes less than 300 and exponentiate a 60 digit number to the 6th power
go	28553	game program from SPEC benchmarks	skill level 10, board size 13
bison	10179	LR(1) parser generator	parser for flex input
jpeg	19842	graphics compression code from SPEC	86K gif file
gzip	7331	file compression program	inflate.c portion of gzip code

Figure 5.1 Program descriptions

Program	Heap Model		
	SMALL	MEDIUM	LARGE
tsp	4	4	x
mlink	4	3	x
fft	4	4	x
clean	4	4	1
cachesim	4	4	x
dhrystone	4	4	x
water	4	4	x
indent	0	0	0
allroots	4	4	x
bc	4	3	1
go	4	4	x
bison	4	4	1
jpeg	0	0	x
gzip	4	4	1

Figure 5.2 Experiment chart

program without MOD/REF analysis. These three columns are each followed by columns of percentages showing the memory, time, or SSA names in the version without MOD/REF analysis over the version with MOD/REF analysis. This experiment shows that MOD/REF analysis is clearly beneficial, reducing the memory, time, and number of SSA names by multiple factors. Particularly important is the reduction in memory required. Three programs required over 100M just to generate the SSA form when no MOD/REF analysis was done. These versions of the programs would have been difficult or impossible to analyze due to space considerations. This is why we stopped this experiment after building the SSA form instead of continuing on with the complete analysis. Note that we perform MOD/REF analysis not only to make pointer analysis feasible, but also to improve its accuracy. See Section 4.4.2 and Figure 4.12 for how MOD/REF analysis can improve the accuracy of pointer analysis.

5.1.2 Multiple representations

We also tested the usefulness of multiple representations. We used a linked-list and a vector set to represent our sets of pointer values in the versions of our pointer analyzer that used no path expressions. The sets of pointer values were initially represented

Program	Memory		Time		SSA names	
	(M)	$\% \frac{w/oMR}{w/MR}$	(s)	$\% \frac{w/oMR}{w/MR}$		$\% \frac{w/oMR}{w/MR}$
tsp	1.9	173.8	0.34	283.3	23079	340.0
mlink	177.2	250.9	42.52	328.1	2021324	442.1
fft	1.9	240.2	0.38	316.7	23966	432.7
clean	85.5	247.9	19.01	320.6	1179195	745.4
cachesim	23.5	215.3	4.84	264.5	365071	560.5
dhrystone	0.8	175.0	0.12	200.0	11792	445.1
water	8.0	410.0	1.63	652.0	107159	947.2
indent	60.7	238.8	13.94	282.8	767176	460.4
allroots	0.4	175.4	0.07	233.3	4237	405.8
bc	42.3	240.2	9.56	295.1	571839	614.6
go	226.6	185.4	65.85	212.5	3057220	234.2
bison	108.3	385.5	26.79	588.8	1485962	1393.6
jpeg	83.6	183.1	18.28	223.2	1067979	336.6
gzip	45.7	278.8	10.88	355.6	631979	501.3

Figure 5.3 Pointer analysis without MOD/REF analysis results

with a linked-list. These sets were converted to a denser vector set representation when the set size passed a threshold of ten. To test the effectiveness of using multiple representations we created a version of the analyzer that only used a linked-list representation. This was done by setting the threshold to 1000. Since the maximum set size generated by our test-suite using the MEDIUM heap model was 178, this ensured that only a linked-list representation was used. The results comparing the versions of the analyzer with different thresholds are shown in Figure 5.4. Note that for these results, the data structures used to convert between the two formats were still created, although we knew they would not be used. Also, the timing results only include the time necessary to perform the propagation of pointer values. It does not include the time to convert the program into SSA form, to perform MOD/REF analysis, and to perform annotation. The first two phases are not affected by the representation used for our pointer sets, and the last phase is only modestly affected.

The results show that multiple representations have a varied impact on propagation time. The reason that multiple representations slowed down propagation for some programs may be due to the cost of converting to the dense representation. On the other hand, the reason that propagation was faster for some programs may have been due to the lower memory requirements of the dense representation. Multiple representations had little impact on memory requirements except for **indent** and **jpeg**. This is to be expected since these two programs have the highest average pointer set size. The denser representation is only beneficial when large sets need to be represented. Without multiple representations, pointer analysis would not have been possible on **jpeg**. This was the main benefit of using multiple representations; it made the analysis of **jpeg** feasible.

In many of our test cases, there is a reduction in memory used when going from no path expressions to path expressions with a path length of one (see Figures 5.5 and 5.6). Our results in the case of no path expressions were generated using multiple representations. The results with path expressions were generated with an exclusively linked-list representation. Since our purpose in using multiple representations is to reduce memory requirements, it might be asked why multiple representations are used if a single representation with the additional memory demands of path expressions frequently uses less memory. To answer this question it should be noted that multiple representations are only beneficial if significant numbers of SSA names are converted to the denser vector set representation. This conversion only occurs at the threshold size of ten. Only four programs, **clean**, **indent**, **bc**, and **jpeg**, had an average set size greater than one. Thus we should only expect multiple representations to be beneficial in these cases. For our test suite, our motivation for using multiple representations was to make analysis of **jpeg** feasible. Without this test case, we would not have needed multiple representations.

5.1.3 Analyzer varying by heap model and path expressions

We organize our results in this section by dividing it into three parts, one for each heap model. We start with the MEDIUM heap model because we consider it our baseline model.

5.1.3.1 MEDIUM heap model

Figures 5.5 and 5.6 show the time and space needed to completely analyze the programs with a MEDIUM heap model. Various statistics are also shown. By "completely analyze", we mean MOD/REF analysis, generation of SSA form, propagation, and annotation. We attempted to analyze each program using path expressions with a length from zero to four. For **mlink**, **indent**, **bc**, and **jpeg** only the shorter lengths were feasible due to time and/or space constraints. When doing analysis with no path expressions (*i.e.*, a path length of zero), a linked-list was used to represent the pointer sets of sparse SSA names. This representation was converted into a vector set when the size grew beyond a threshold of ten. Analysis with path expressions

Program	Memory (M)			Time (s)		
Threshold	1000	10	$\% \frac{threshold1000}{threshold10}$	1000	10	$\% \frac{threshold1000}{threshold10}$
tsp	1.4	1.3	102.2	0.05	0.05	100.0
mlink	93.7	93.7	100.0	15.53	15.90	97.7
fft	1.3	1.3	102.1	0.03	0.06	50.0
clean	39.0	39.0	99.9	9.24	9.98	92.6
cachesim	14.5	14.5	100.2	0.87	0.83	104.8
dhrystone	0.8	0.7	104.4	0.02	0.03	66.7
water	2.6	2.5	101.2	0.08	0.09	88.9
indent	100.0	35.4	282.6	54.81	36.06	152.0
allroots	0.4	0.3	109.5	0.03	0.00	*
bc	22.4	22.2	101.0	12.80	25.27	50.7
go	143.8	143.7	100.0	4.63	5.10	90.8
bison	30.9	30.9	100.0	1.71	1.58	108.2
jpeg	*	91.0	*	*	11393.66	*
gzip	21.3	21.3	100.0	1.34	1.37	97.8

Figure 5.4 Comparison between multiple and single representations

required a linked-list representation. This difference in representation should be remembered when comparing the results for various path lengths. The memory column shows the maximum amount of memory used in analyzing the program. It does not include the time needed to parse all the files. Since memory requirements were a major concern we chose to build the pruned-SSA form. The regions column shows the number of memory regions in the program. Thus, it shows the number of possible pointer values. Figures 5.7 and 5.8 show various statistics for the size of the pointer sets for all SSA names ("all SSA names" columns) and for the SSA names of base registers of pointer-based memory operations ("base registers of pt. based memop" columns). The "Max" column in the first group ("all SSA names") shows the size of the largest pointer set for all SSA names. Likewise, the "Max" column in the second group ("base registers of pt. based memop" columns) shows the size of the largest pointer set for all SSA names that are base registers of a pointer-based memory operation. The "Avg" columns show the average size of the pointer sets for each group of SSA names. The "0" column shows the number of SSA names that have zero pointers. The "< 10" column shows the number of SSA names with less than ten pointers. The "Num" column on the far right is the number of pointer-based memory operations in the program. In Figures 5.7 and 5.8 and for all other figures on pointer results, we do not have a column showing the minimum number of pointer values in a pointer-based memory operation since this value was always one for all programs and for all forms of our pointer analyzer. Note that for analysis with path expressions, we count a pointer value once even if it appears with multiple path expressions.

As was expected, increasing path length usually increased the time and space requirements for analysis. However, for some cases increasing the path length reduced the time requirements significantly. For example, there is a time reduction in the **mlink** test case when going from a path length of two to three. This result is somewhat surprising, but it is not unexplainable. The increased precision from longer path expressions can remove some extraneous propagation due to false returns and pointer mixing. This reduction in propagation can be greater than the cost of longer path expressions. We have not been able to verify that this is the main reason for the reduction in time when moving to longer path expressions (*i.e.*, we have not been able to find the exact piece of code where extraneous propagation is removed). One piece of evidence that this is the case is the lower average number of pointer values for all SSA names. This reduction must be due to the increased length of the path expressions.

In many of the cases, there is a reduction in memory used when going from no path expressions to a path length of one. This is due primarily to the fact that our pointer sets had multiple formats when the path length was zero, but they had a single format when the path length was greater. A single format allowed some data structures to be eliminated that were necessary for conversion when using multiple formats. In particular, it should be noted that this memory reduction is not due to the increased accuracy obtained by using path expressions.

The program **indent** is difficult to analyze because of a large compiler-initialized array of structures, `pro`, which is used to handle formatting options. Each of these structures has a pointer value. Since we do not do any array section analysis, any load from this array will place all the pointer values in it into the target register. This creates a long list of pointers that is widely propagated. The program **jpeg** is difficult to analyze because of a large number, 278, of indirect procedure calls.

5.1.3.2 SMALL heap model

Figures 5.9, 5.10, 5.11, and 5.12, show the performance and pointer results of the analyzer with a SMALL heap model. The ratios of these results to our baseline analysis are shown in Figures 5.13, 5.14, 5.15, and 5.16. A glance at the "Regions" column shows that the number of regions is usually reduced by a few percent by using a SMALL heap model. In the **go** test case, there are no regions created to model the heap in the MEDIUM heap model, since there are no heap allocation call

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
tsp	0	1.3	0.24	6788	66
	1	1.4	0.36		
	2	1.4	0.33		
	3	1.4	0.37		
	4	1.4	0.37		
mlink	0	93.7	32.89	457240	651
	1	87.7	511.28		
	2	95.7	8395.17		
	3	108.1	7110.13		
fft	0	1.3	0.23	5539	84
	1	1.2	0.24		
	2	1.2	0.27		
	3	1.2	0.27		
	4	1.2	0.25		
clean	0	39.0	21.77	158197	389
	1	43.9	1060.94		
	2	51.9	3867.86		
	3	64.0	9611.32		
	4	75.0	12400.80		
cachesim	0	14.5	3.66	65129	262
	1	13.6	9.15		
	2	14.1	14.82		
	3	15.0	45.62		
	4	15.9	69.78		
dhrystone	0	0.7	0.12	2649	46
	1	0.6	0.20		
	2	0.6	0.19		
	3	0.6	0.19		
	4	0.6	0.22		

Figure 5.5 Pointer analysis: baseline performance results

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
water	0	2.5	0.43	11313	274
	1	2.5	0.44		
	2	2.5	0.42		
	3	2.5	0.45		
	4	2.5	0.52		
indent	0	35.4	42.94	166640	312
allroots	0	0.3	0.04	1044	31
	1	0.3	0.07		
	2	0.3	0.06		
	3	0.3	0.08		
	4	0.3	0.06		
bc	0	22.2	30.58	93045	282
	1	33.1	1086.86		
	2	51.5	9886.79		
	3	79.0	45680.64		
go	0	143.7	46.64	1305321	817
	1	141.0	136.42		
	2	142.9	956.80		
	3	145.9	5914.16		
	4	149.2	17109.33		
bison	0	30.9	8.59	106624	355
	1	34.0	108.31		
	2	38.0	366.45		
	3	45.1	1267.39		
	4	59.7	4300.30		
jpeg	0	91.0	11539.52	317315	378
gzip	0	21.3	5.86	126063	437
	1	21.3	12.20		
	2	22.0	26.98		
	3	23.7	163.91		
	4	26.4	629.75		

Figure 5.6 Pointer analysis: baseline performance results (cont.)

Program	Path Length	all SSA names Max	0	<10	Avg	base registers of pt. based memop Avg	Max	Num
tsp	0	5	5214	6788	0.2364	1.0602	5	216
	1	5	5214	6788	0.2364	1.0602	5	
	2	5	5214	6788	0.2364	1.0602	5	
	3	5	5214	6788	0.2364	1.0602	5	
	4	5	5214	6788	0.2364	1.0602	5	
mlink	0	38	384653	456232	0.7751	3.1108	38	3394
	1	37	387295	457232	0.4205	2.3724	37	
	2	37	390965	457232	0.3992	2.3724	37	
	3	34	396859	457232	0.3457	2.3385	34	
fft	0	5	4638	5539	0.2083	2.1165	5	103
	1	5	4638	5539	0.2083	2.1165	5	
	2	5	4638	5539	0.2083	2.1165	5	
	3	5	4638	5539	0.2083	2.1165	5	
	4	5	4638	5539	0.2083	2.1165	5	
clean	0	13	126739	158185	1.2157	7.0088	13	1254
	1	13	126739	158185	1.2102	6.8198	13	
	2	13	126788	158185	1.2001	6.7249	13	
	3	13	126917	158193	1.1827	6.6093	13	
	4	13	127242	158193	1.1411	6.5255	13	
cache-sim	0	8	56344	65129	0.2975	1.3723	8	419
	1	8	56747	65129	0.2776	1.3723	8	
	2	8	56981	65129	0.2668	1.3723	8	
	3	8	56981	65129	0.2668	1.3723	8	
	4	8	56981	65129	0.2668	1.3723	8	
dhry-stone	0	4	1834	2649	0.3220	1.2712	4	59
	1	4	2078	2649	0.2269	1.2712	4	
	2	4	2078	2649	0.2269	1.2712	4	
	3	4	2078	2649	0.2269	1.2712	4	
	4	4	2078	2649	0.2269	1.2712	4	

Figure 5.7 Pointer analysis: baseline pointer results

Program	Path Length	all SSA names				base registers of pt. based memop		
		Max	0	<10	Avg	Avg	Max	Num
water	0	8	9022	11313	0.2377	1.5346	8	260
	1	8	9022	11313	0.2377	1.5346	8	
	2	8	9022	11313	0.2377	1.5346	8	
	3	8	9022	11313	0.2377	1.5346	8	
	4	8	9022	11313	0.2377	1.5346	8	
indent	0	114	52615	70697	63.4176	12.4430	114	1465
allroots	0	2	880	1044	0.1925	1.5714	2	35
	1	2	880	1044	0.1925	1.5714	2	
	2	2	880	1044	0.1925	1.5714	2	
	3	2	880	1044	0.1925	1.5714	2	
	4	2	880	1044	0.1925	1.5714	2	
bc	0	34	48274	55413	6.1801	10.8604	34	1067
	1	31	48843	64597	3.8592	8.3280	31	
	2	31	50014	65761	3.7090	8.3074	31	
	3	31	50052	67729	3.4759	8.3074	31	
go	0	102	1256788	1305285	0.0386	4.2576	102	493
	1	102	1256962	1305285	0.0384	4.2576	102	
	2	102	1256962	1305285	0.0384	4.2576	102	
	3	102	1256962	1305285	0.0384	4.2576	102	
	4	102	1256962	1305285	0.0384	4.2576	102	
bison	0	14	75158	106619	0.8793	1.3199	14	2166
	1	14	75158	106619	0.8793	1.3199	14	
	2	14	75158	106619	0.8793	1.3199	14	
	3	14	75158	106619	0.8793	1.3199	14	
	4	14	75158	106619	0.8793	1.3199	14	
jpeg	0	178	106397	110674	108.2039	76.4131	178	3430
gzip	0	10	103822	126054	0.3138	2.3671	10	425
	1	9	103887	126063	0.2263	1.8329	9	
	2	9	105251	126063	0.2132	1.8047	9	
	3	9	105592	126063	0.2105	1.8047	9	
	4	9	105592	126063	0.2105	1.8047	9	

Figure 5.8 Pointer analysis: baseline pointer results (cont.)

sites to library functions. When a single name was used to represent the heap in the SMALL heap model, this name replaced zero regions, thus increasing the total number of regions by one. Using the SMALL heap model allowed us to analyze **mlink** and **bc** with path expressions of length four. This was not possible with the MEDIUM heap model. The SMALL heap model usually required less time and memory than the MEDIUM heap model. On some cases, **mlink** and **bc**, the SMALL heap model required substantially less time. This can be explained since we have fewer pointer values to propagate. Also, fewer SSA names need to be created. Remember, heap names are addressed global regions. Addressed global regions are the most expensive regions to analyze, since they must be placed in the referenced and modified lists of all pointer-based memory operations in order to create the SSA form of the program. On the other hand, using a SMALL heap model can also use more time and memory than the same analysis with a MEDIUM heap model. This can be explained since a single heap name causes more approximation in interactions with the heap. These approximations will cause more unnecessary pointers to be propagated which will increase the time and space requirements of the analysis.

5.1.3.3 LARGE heap model

Figures 5.17 and 5.18 show the performance and performance results of the analyzer with a LARGE heap model. Figures 5.19 and 5.20 show the ratios of these results to our baseline analysis. The "Regions" column shows that using a LARGE heap model as opposed to a MEDIUM heap model increases the number of regions from 0.2% to over 30%. This increase in addressed global regions has a modest impact on the amount of space required but a substantial impact on the time required for analysis. Unlike the jump from a SMALL heap model to a a MEDIUM heap model, the jump from a MEDIUM heap model to a LARGE heap model always takes more time.

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
tsp	0	1.4	0.30	6747	66
	1	1.4	0.32		
	2	1.4	0.36		
	3	1.4	0.35		
	4	1.4	0.35		
mlink	0	85.8	19.32	412535	621
	1	76.8	56.70		
	2	79.1	315.78		
	3	84.1	847.50		
	4	98.2	18738.31		
fft	0	1.0	0.20	4766	75
	1	0.9	0.19		
	2	1.0	0.26		
	3	1.0	0.24		
	4	1.0	0.23		
clean	0	38.4	19.47	155973	387
	1	42.5	1011.80		
	2	48.5	3633.67		
	3	56.8	7767.14		
	4	64.5	10898.62		
cachesim	0	13.8	3.56	60796	252
	1	13.0	10.04		
	2	13.2	14.70		
	3	13.9	56.96		
	4	14.4	76.15		
dhrystone	0	0.7	0.13	2581	45
	1	0.6	0.19		
	2	0.6	0.16		
	3	0.6	0.19		
	4	0.6	0.17		

Figure 5.9 SMALL heap model performance results

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
water	0	2.5	0.40	11176	273
	1	2.4	0.42		
	2	2.5	0.49		
	3	2.5	0.48		
	4	2.5	0.47		
indent	0	35.4	43.42	166479	312
allroots	0	0.4	0.04	1044	31
	1	0.3	0.04		
	2	0.3	0.05		
	3	0.3	0.07		
	4	0.3	0.07		
bc	0	20.0	11.63	84564	274
	1	27.1	537.99		
	2	37.0	4559.80		
	3	52.2	18166.53		
	4	73.2	68545.03		
go	0	144.1	47.03	1308877	818
	1	141.3	139.21		
	2	143.2	1013.65		
	3	146.2	6161.38		
	4	149.5	17805.91		
bison	0	30.7	8.47	104816	354
	1	33.8	118.65		
	2	37.8	420.34		
	3	44.8	1416.65		
	4	59.5	4823.35		
jpeg	0	89.6	10769.38	314948	377
gzip	0	21.3	6.21	124582	435
	1	21.2	13.57		
	2	22.0	28.90		
	3	24.0	182.60		
	4	27.1	706.59		

Figure 5.10 SMALL heap model performance results (cont.)

Program	Path Length	all SSA names				base registers of pt. based memop		
		Max	0	<10	Avg	Avg	Max	Num
tsp	0	5	5204	6747	0.2333	1.0602	5	216
	1	5	5204	6747	0.2333	1.0602	5	
	2	5	5204	6747	0.2333	1.0602	5	
	3	5	5204	6747	0.2333	1.0602	5	
	4	5	5204	6747	0.2333	1.0602	5	
mlink	0	28	354145	412533	0.1558	1.1759	28	3394
	1	28	355311	412533	0.1526	1.1759	28	
	2	28	358981	412533	0.1425	1.1759	28	
	3	28	361709	412533	0.1359	1.1759	28	
	4	28	361709	412533	0.1359	1.1759	28	
fft	0	2	3886	4766	0.1861	1.0680	2	103
	1	2	3886	4766	0.1861	1.0680	2	
	2	2	3886	4766	0.1861	1.0680	2	
	3	2	3886	4766	0.1861	1.0680	2	
	4	2	3886	4766	0.1861	1.0680	2	
clean	0	11	126689	155962	0.8610	5.3102	11	1254
	1	11	126689	155963	0.8576	5.1970	11	
	2	11	126738	155963	0.8505	5.1292	11	
	3	11	126867	155971	0.8388	5.0518	11	
	4	11	127192	155971	0.8129	5.0016	11	
cachesim	0	6	48648	60796	0.3266	1.2905	6	420
	1	6	49922	60796	0.2855	1.2905	6	
	2	6	52487	60796	0.2024	1.2905	6	
	3	6	52487	60796	0.2024	1.2905	6	
	4	6	52487	60796	0.2024	1.2905	6	
dhrystone	0	3	1830	2581	0.3003	1.2034	3	59
	1	3	2062	2581	0.2073	1.2034	3	
	2	3	2062	2581	0.2073	1.2034	3	
	3	3	2062	2581	0.2073	1.2034	3	
	4	3	2062	2581	0.2073	1.2034	3	

Figure 5.11 SMALL heap model pointer results

Program	Path Length	all SSA names				base registers of pt. based memop		
		Max	0	<10	Avg	Avg	Max	Num
water	0	8	8906	11176	0.2387	1.5346	8	260
	1	8	8906	11176	0.2387	1.5346	8	
	2	8	8906	11176	0.2387	1.5346	8	
	3	8	8906	11176	0.2387	1.5346	8	
	4	8	8906	11176	0.2387	1.5346	8	
indent	0	114	52614	70561	63.4616	12.4430	114	1465
allroots	0	2	880	1044	0.1925	1.5714	2	35
	1	2	880	1044	0.1925	1.5714	2	
	2	2	880	1044	0.1925	1.5714	2	
	3	2	880	1044	0.1925	1.5714	2	
	4	2	880	1044	0.1925	1.5714	2	
bc	0	27	48225	84531	2.9562	5.8866	27	1067
	1	27	48714	84531	2.1943	5.8472	27	
	2	27	49877	84531	2.0846	5.8341	27	
	3	27	49907	84531	1.9217	5.8341	27	
	4	27	50328	84531	1.9158	5.8341	27	
go	0	102	1260367	1308841	0.0384	4.2576	102	493
	1	102	1260541	1308841	0.0383	4.2576	102	
	2	102	1260541	1308841	0.0383	4.2576	102	
	3	102	1260541	1308841	0.0383	4.2576	102	
	4	102	1260541	1308841	0.0383	4.2576	102	
bison	0	14	73374	104811	0.9108	1.3199	14	2166
	1	14	73374	104811	0.9108	1.3199	14	
	2	14	73374	104811	0.9108	1.3199	14	
	3	14	73374	104811	0.9108	1.3199	14	
	4	14	73374	104811	0.9108	1.3199	14	
jpeg	0	177	106378	110652	107.1176	75.5703	177	3430
gzip	0	10	100455	124574	0.4651	2.5929	10	425
	1	9	101413	124582	0.2515	1.8706	9	
	2	9	102778	124582	0.2382	1.8424	9	
	3	9	103119	124582	0.2354	1.8424	9	
	4	9	103119	124582	0.2354	1.8424	9	

Figure 5.12 SMALL heap model pointer results (cont.)

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
tsp	0	102.2	125.0	99.4	100.0
	1	99.7	88.9		
	2	99.7	109.1		
	3	99.7	94.6		
	4	99.7	94.6		
mlink	0	91.6	58.7	90.2	95.4
	1	87.5	11.1		
	2	82.7	3.8		
	3	77.8	11.9		
	4	*	*		
fft	0	73.4	87.0	86.0	89.3
	1	80.2	79.2		
	2	83.3	96.3		
	3	83.4	88.9		
	4	83.5	92.0		
clean	0	98.5	89.4	98.6	99.5
	1	96.9	95.4		
	2	93.5	93.9		
	3	88.7	80.8		
	4	86.0	87.9		
cachesim	0	95.1	97.3	93.3	96.2
	1	95.7	109.7		
	2	94.0	99.2		
	3	93.1	124.9		
	4	90.9	109.1		
dhrystone	0	98.8	108.3	97.4	97.8
	1	97.7	95.0		
	2	97.6	84.2		
	3	97.6	100.0		
	4	97.5	77.3		

Figure 5.13 SMALL/baseline heap model performance percentages

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
water	0	99.4	93.0	98.8	99.6
	1	98.9	95.5		
	2	98.9	116.7		
	3	98.9	106.7		
	4	98.9	90.4		
indent	0	100.0	101.1	99.9	100.0
allroots	0	109.5	100.0	100.0	100.0
	1	100.0	57.1		
	2	100.0	83.3		
	3	100.0	87.5		
	4	100.0	116.7		
bc	0	90.2	38.0	90.9	97.2
	1	81.7	49.5		
	2	71.9	46.1		
	3	66.0	39.8		
	4	*	*		
go	0	100.3	100.8	100.3	100.1
	1	100.2	102.0		
	2	100.2	105.9		
	3	100.2	104.2		
	4	100.2	104.1		
bison	0	99.4	98.6	98.3	99.7
	1	99.5	109.5		
	2	99.6	114.7		
	3	99.3	111.8		
	4	99.7	112.2		
jpeg	0	98.5	93.3	99.3	99.7
gzip	0	99.7	106.0	98.8	99.5
	1	99.4	111.2		
	2	99.9	107.1		
	3	101.0	111.4		
	4	102.5	112.2		

Figure 5.14 SMALL/baseline heap model performance percentages (cont.)

Program	Path Length	all SSA names Max	0	<10	Avg	base registers of pt. based memop Avg	Max
tsp	0	100.0	99.8	99.4	98.7	100.0	100.0
	1	100.0	99.8	99.4	98.7	100.0	100.0
	2	100.0	99.8	99.4	98.7	100.0	100.0
	3	100.0	99.8	99.4	98.7	100.0	100.0
	4	100.0	99.8	99.4	98.7	100.0	100.0
mlink	0	73.7	92.1	90.4	20.1	37.8	73.7
	1	75.7	91.7	90.2	36.3	49.6	75.7
	2	75.7	91.8	90.2	35.7	49.6	75.7
	3	82.4	91.1	90.2	39.3	50.3	82.4
	4	*	*	*	*	*	*
fft	0	40.0	83.8	86.0	89.3	50.5	40.0
	1	40.0	83.8	86.0	89.3	50.5	40.0
	2	40.0	83.8	86.0	89.3	50.5	40.0
	3	40.0	83.8	86.0	89.3	50.5	40.0
	4	40.0	83.8	86.0	89.3	50.5	40.0
clean	0	84.6	100.0	98.6	70.8	75.8	84.6
	1	84.6	100.0	98.6	70.9	76.2	84.6
	2	84.6	100.0	98.6	70.9	76.3	84.6
	3	84.6	100.0	98.6	70.9	76.4	84.6
	4	84.6	100.0	98.6	71.2	76.6	84.6
cachesim	0	75.0	86.3	93.3	109.8	94.0	75.0
	1	75.0	88.0	93.3	102.8	94.0	75.0
	2	75.0	92.1	93.3	75.9	94.0	75.0
	3	75.0	92.1	93.3	75.9	94.0	75.0
	4	75.0	92.1	93.3	75.9	94.0	75.0
dhrystone	0	75.0	99.8	97.4	93.3	94.7	75.0
	1	75.0	99.2	97.4	91.4	94.7	75.0
	2	75.0	99.2	97.4	91.4	94.7	75.0
	3	75.0	99.2	97.4	91.4	94.7	75.0
	4	75.0	99.2	97.4	91.4	94.7	75.0

Figure 5.15 SMALL/baseline heap model pointer statistic percentages

Program	Path Length	all SSA names				base registers of pt. based memop	
		Max	0	<10	Avg	Avg	Max
water	0	100.0	98.7	98.8	100.4	100.0	100.0
	1	100.0	98.7	98.8	100.4	100.0	100.0
	2	100.0	98.7	98.8	100.4	100.0	100.0
	3	100.0	98.7	98.8	100.4	100.0	100.0
	4	100.0	98.7	98.8	100.4	100.0	100.0
indent	0	100.0	100.0	99.8	100.1	100.0	100.0
allroots	0	100.0	100.0	100.0	100.0	100.0	100.0
	1	100.0	100.0	100.0	100.0	100.0	100.0
	2	100.0	100.0	100.0	100.0	100.0	100.0
	3	100.0	100.0	100.0	100.0	100.0	100.0
	4	100.0	100.0	100.0	100.0	100.0	100.0
bc	0	79.4	99.9	152.5	47.8	54.2	79.4
	1	87.1	99.7	130.9	56.9	70.2	87.1
	2	87.1	99.7	128.5	56.2	70.2	87.1
	3	87.1	99.7	124.8	55.3	70.2	87.1
	4	*	*	*	*	*	*
go	0	100.0	100.3	100.3	99.5	100.0	100.0
	1	100.0	100.3	100.3	99.7	100.0	100.0
	2	100.0	100.3	100.3	99.7	100.0	100.0
	3	100.0	100.3	100.3	99.7	100.0	100.0
	4	100.0	100.3	100.3	99.7	100.0	100.0
bison	0	100.0	97.6	98.3	103.6	100.0	100.0
	1	100.0	97.6	98.3	103.6	100.0	100.0
	2	100.0	97.6	98.3	103.6	100.0	100.0
	3	100.0	97.6	98.3	103.6	100.0	100.0
	4	100.0	97.6	98.3	103.6	100.0	100.0
jpeg	0	99.4	100.0	100.0	99.0	98.9	99.4
gzip	0	100.0	96.8	98.8	148.2	109.5	100.0
	1	100.0	97.6	98.8	111.1	102.1	100.0
	2	100.0	97.7	98.8	111.7	102.1	100.0
	3	100.0	97.7	98.8	111.8	102.1	100.0
	4	100.0	97.7	98.8	111.8	102.1	100.0

Figure 5.16 SMALL/baseline heap model pointer statistic percentages (cont.)

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
clean	0	51.6	719.65	222499	449
	1	95.3	14198.83		
indent	0	40.7	151.52	189888	339
bc	0	26.4	156.94	118917	307
	1	44.5	1509.67		
bison	0	57.8	24.27	319591	475
	1	64.7	1265.62		
gzip	0	21.4	6.14	126460	438
	1	21.3	12.62		

Figure 5.17 LARGE heap model performance results

Program	Path Length	all SSA names				base registers of pt. based memop		
		Max	0	<10	Avg	Avg	Max	Num
clean	0	46	130632	164014	10.9881	22.2504	46	1254
	1	46	131693	165099	10.5285	21.6156	46	
indent	0	136	58237	76184	77.7312	14.4273	136	1465
bc	0	48	53612	60748	14.3472	19.6626	48	1067
	1	40	54381	70152	5.6674	8.8013	30	
bison	0	94	199714	310688	1.3528	1.9224	26	2166
	1	94	201450	310688	1.3147	1.9224	26	
gzip	0	10	104427	126451	0.3111	2.3671	10	425
	1	9	104492	126460	0.2240	1.8329	9	

Figure 5.18 LARGE heap model pointer results

Program	Path Length	Memory (M)	Time (s)	SSA Names	Regions
clean	0	132.3	3305.7	140.6	115.4
	1	217.1	1338.3		
indent	0	114.8	352.9	114.0	108.7
bc	0	119.0	513.2	127.8	108.9
	1	134.3	138.9		
bison	0	187.2	282.5	299.7	133.8
	1	190.4	1168.5		
gzip	0	100.3	104.8	100.3	100.2
	1	100.2	103.4		

Figure 5.19 LARGE/baseline heap model performance percentages

Program	Path Length	all SSA names				base registers of pt. based memop	
		Max	0	<10	Avg	Avg	Max
clean	0	353.8	103.1	103.7	903.8	317.5	353.8
	1	353.8	103.9	104.4	870.0	317.0	353.8
indent	0	119.3	110.7	107.8	122.6	115.9	119.3
bc	0	141.2	111.1	109.6	232.2	181.0	141.2
	1	129.0	111.3	108.6	146.9	105.7	96.8
bison	0	671.4	265.7	291.4	153.8	145.6	185.7
	1	671.4	268.0	291.4	149.5	145.6	185.7
gzip	0	100.0	100.6	100.3	99.1	100.0	100.0
	1	100.0	100.6	100.3	99.0	100.0	100.0

Figure 5.20 LARGE/baseline heap model pointer percentages

5.2 Analyzed code performance

The code produced by our pointer analyzer was tested by running it through our compiler environment. Our environment consists of a number of optimization passes that take in `ILOC` and produce optimized `ILOC`. By structuring our optimizer as a sequence of independent filters, we gain flexibility in our testing procedures. The optimizations used in this section were:

- **RP**: register promotion
- **LP**: loop peeling
- **VN**: value numbering
- **LCM**: lazy code motion
- **CP**: constant propagation
- **LICM**: loop-invariant code motion
- **DCE**: dead code elimination
- **CO**: coalesce
- **RA**: register allocation
- **CL**: clean

The specific sequence of optimization passes used in this section was:

RP | **LP** | **VN** | **LCM** | **RP** | **CP** | **LICM** | **DCE** | **VN** | **LCM** | **LICM** | **VN** | **LCM** | **CP** | **DCE** | **CO** | **RA** | **CL**

After the code was optimized, it was translated into an equivalent C program. This C code was instrumented to record the number of `ILOC` operations, `ILOC` loads, and `ILOC` stores executed. Once this instrumented code was executed, the results were checked against the output produced by the original C source code when compiled with a UNIX C compiler.

In our tests of the analyzed programs, two parameters for pointer analysis were investigated, type of heap model and type of path information (up to length four). The results for the pointer analyzed versions for the programs were compared against versions with no analysis and versions with MOD/REF analysis. Some programs could not be tested with lengthy path expressions. Two programs, **jpeg** and **indent**, could not be tested with any path expressions. Only five programs (**clean**, **indent**, **bc**, **bison**, and **gzip**) performed their own heap allocation. We have split up the results in this section into four subsections. The first subsection presents the results obtained with no pointer analysis. The last three subsections present the results obtained by by using various heap models.

5.2.1 No pointer analysis

In order to compare the effectiveness of our pointer analyses, we generated versions of each program with no analysis and with MOD/REF analysis. Figure 5.21 shows the results in terms of total operations executed. Figures 5.22 and 5.23, show number of stores and loads executed, respectively. In each figure we show the absolute number of operations/stores/loads[8] executed for each of the fourteen test programs.

5.2.2 MEDIUM heap model

We present the results of the analyzer with a MEDIUM heap model before the results obtained with other heap models, because the MEDIUM heap model is our baseline model (see Appendix B for additional results with this model). Figures 5.24, 5.26, and 5.28 show the results obtained by using this model. For each program tested, we varied the path expressions used in the analysis. Path expressions with path lengths varying from zero to four were used. In Figures 5.25, 5.27, and 5.29, we show a comparison of the various versions of each program (including versions with no analysis and MOD/REF analysis) with the version produced with a MEDIUM heap model and no path expressions. The MEDIUM heap model with no path expressions is our $base\_analysis$.

The formula for these comparisons is:

$$\frac{100*(other\_analysis - base\_analysis)}{other\_analysis}$$

These figures show that pointer analysis can produce substantial improvements over no analysis and MOD/REF analysis. Path expressions offered very little improvement over analysis without path expressions. Most of the improvements obtained by path expressions could be obtained from path expressions of length one. These improvements due to pointer analysis were concentrated in our load results. This is to be expected since pointer analysis deals with memory and thus should improve our

[8] From now on we will just refer to operations even if we also mean stores or loads.

Program	Original	MOD/REF
tsp	729921	657271
mlink	134483827	126772024
fft	13634058	12635914
clean	1141512	1099823
cachesim	11960530	11589109
dhrystone	596198	582185
water	14798643	13476586
indent	945899	867212
allroots	1028	1010
bc	5598537	5561348
go	501977911	496832956
bison	3863186	3334118
jpeg	36904868	36963318
gzip(enc)	5736537	5711089
gzip(dec)	979824	976879

Figure 5.21 No pointer analysis: total operations

Program	Original	MOD/REF
tsp	51047	51048
mlink	2507703	2505951
fft	1036396	1036398
clean	85022	82442
cachesim	685931	594390
dhrystone	60007	60010
water	1035129	1035175
indent	72615	69140
allroots	11	11
bc	243650	238488
go	20357668	20203423
bison	539944	539606
jpeg	2503016	2503022
gzip(enc)	216990	212300
gzip(dec)	17768	17379

Figure 5.22 No pointer analysis: store operations

Program	Original	MOD/REF
tsp	166549	113721
mlink	32004568	27044130
fft	2012056	1252763
clean	207510	183382
cachesim	2102248	1901536
dhrystone	68033	62021
water	3060022	2386374
indent	206751	147453
allroots	163	145
bc	845705	817587
go	93336622	89224694
bison	942296	553731
jpeg	6535643	6596102
gzip(enc)	901537	854592
gzip(dec)	138323	135968

Figure 5.23 No pointer analysis: load operations

Program	Path 0	Path 1	Path 2	Path 3	Path 4
tsp	651487	651487	651487	651487	651487
mlink	124425317	124424420	124423998	124423998	
fft	12550636	12550618	12550618	12550618	12550618
clean	1108473	1105261	1105261	1105261	1105261
cachesim	11587501	11587501	11587501	11587501	11587501
dhrystone	552181	548186	548186	548186	548186
water	12726871	12726871	12726871	12726871	12726871
indent	867180				
allroots	1000	1000	1000	1000	1000
bc	5550821	5546264	5546723	5546723	
go	486937168	486777895	486872714	486872714	486872714
bison	3331125	3327915	3327915	3327915	3327915
jpeg	36962001				
gzip(enc)	5678893	5678893	5678893	5678893	5678893
gzip(dec)	976366	976366	976366	976366	976366

Figure 5.24 MEDIUM heap model: total operations

Program	% eliminated					
	Original	MOD/REF	Path 1	Path 2	Path 3	Path 4
tsp	10.7	0.9	0.0	0.0	0.0	0.0
mlink	7.5	1.9	-0.0	-0.0	-0.0	
fft	7.9	0.7	-0.0	-0.0	-0.0	-0.0
clean	2.9	-0.8	-0.3	-0.3	-0.3	-0.3
cachesim	3.1	0.0	0.0	0.0	0.0	0.0
dhrystone	7.4	5.2	-0.7	-0.7	-0.7	-0.7
water	14.0	5.6	0.0	0.0	0.0	0.0
indent	8.3	0.0				
allroots	2.7	1.0	0.0	0.0	0.0	0.0
bc	0.9	0.2	-0.1	-0.1	-0.1	
go	3.0	2.0	-0.0	-0.0	-0.0	-0.0
bison	13.8	0.1	-0.1	-0.1	-0.1	-0.1
jpeg	-0.2	0.0				
gzip(enc)	1.0	0.6	0.0	0.0	0.0	0.0
gzip(dec)	0.4	0.1	0.0	0.0	0.0	0.0

Figure 5.25 MEDIUM heap model: total operation removal percentages

Program	Path 0	Path 1	Path 2	Path 3	Path 4
tsp	51048	51048	51048	51048	51048
mlink	2352204	2352200	2352200	2352200	
fft	1003608	1003599	1003599	1003599	1003599
clean	82402	80189	80189	80189	80189
cachesim	594390	594390	594390	594390	594390
dhrystone	56010	56010	56010	56010	56010
water	1069645	1069645	1069645	1069645	1069645
indent	69141				
allroots	11	11	11	11	11
bc	238488	238488	239639	239639	
go	20403336	20412840	20475320	20475320	20475320
bison	539606	539606	539606	539606	539606
jpeg	2503022				
gzip(enc)	198823	198823	198823	198823	198823
gzip(dec)	17233	17233	17233	17233	17233

Figure 5.26 MEDIUM heap model: stores

Program	% eliminated					
	Original	MOD/REF	Path 1	Path 2	Path 3	Path 4
tsp	-0.0	0.0	0.0	0.0	0.0	0.0
mlink	6.2	6.1	-0.0	-0.0	-0.0	
fft	3.2	3.2	-0.0	-0.0	-0.0	-0.0
clean	3.1	0.0	-2.8	-2.8	-2.8	-2.8
cachesim	13.3	0.0	0.0	0.0	0.0	0.0
dhrystone	6.7	6.7	0.0	0.0	0.0	0.0
water	-3.3	-3.3	0.0	0.0	0.0	0.0
indent	4.8	-0.0				
allroots	0.0	0.0	0.0	0.0	0.0	0.0
bc	2.1	0.0	0.0	0.5	0.5	
go	-0.2	-1.0	0.0	0.4	0.4	0.4
bison	0.1	0.0	0.0	0.0	0.0	0.0
jpeg	-0.0	0.0				
gzip(enc)	8.4	6.3	0.0	0.0	0.0	0.0
gzip(dec)	3.0	0.8	0.0	0.0	0.0	0.0

Figure 5.27 MEDIUM heap model: store removal percentages

Program	Path 0	Path 1	Path 2	Path 3	Path 4
tsp	107937	107937	107937	107937	107937
mlink	25517809	25517130	25517019	25517019	
fft	1204103	1204103	1204103	1204103	1204103
clean	180429	179091	179091	179091	179091
cachesim	1900611	1900611	1900611	1900611	1900611
dhrystone	50021	46021	46021	46021	46021
water	1673879	1673879	1673879	1673879	1673879
indent	147735				
allroots	141	141	141	141	141
bc	812012	808606	807914	807914	
go	83945430	83851902	83865727	83865727	83865727
bison	552589	551519	551519	551519	551519
jpeg	6594787				
gzip(enc)	835906	835906	835906	835906	835906
gzip(dec)	135636	135636	135636	135636	135636

Figure 5.28 MEDIUM heap model: loads

Program	% eliminated					
	Original	MOD/REF	Path 1	Path 2	Path 3	Path 4
tsp	35.2	5.1	0.0	0.0	0.0	0.0
mlink	20.3	5.6	-0.0	-0.0	-0.0	
fft	40.2	3.9	0.0	0.0	0.0	0.0
clean	13.1	1.6	-0.7	-0.7	-0.7	-0.7
cachesim	9.6	0.0	0.0	0.0	0.0	0.0
dhrystone	26.5	19.3	-8.7	-8.7	-8.7	-8.7
water	45.3	29.9	0.0	0.0	0.0	0.0
indent	28.5	-0.2				
allroots	13.5	2.8	0.0	0.0	0.0	0.0
bc	4.0	0.7	-0.4	-0.5	-0.5	
go	10.1	5.9	-0.1	-0.1	-0.1	-0.1
bison	41.4	0.2	-0.2	-0.2	-0.2	-0.2
jpeg	-0.9	0.0				
gzip(enc)	7.3	2.2	0.0	0.0	0.0	0.0
gzip(dec)	1.9	0.2	0.0	0.0	0.0	0.0

Figure 5.29 MEDIUM heap model: load removal percentages

memory numbers. It is not clear why stores are not affected as much. It is more beneficial to remove loads than stores since processors must wait for a load to complete before continuing, but stores do not force a processor to pause. Removing load operations will become more and more important as processor speed is increasing more rapidly than memory speed is. Stores on the other hand are cheap since computation does not need to wait on their completion.

We looked at the resulting code to determine the actual mechanism through which pointer analysis improves code. In **water**, we have a very dramatic example (see Figure 5.30). In `CSHIFT`, where most of the program's improvement occurred, we have three arrays that are passed in, `XA`, `XB` and `XL`. Without pointer analysis we do not know if these arrays are non-overlapping. Thus, we must reload all references to these arrays. With pointer analysis, we know that `XL` points to eight stack allocated arrays in functions `POTENG` and `INTERF`, and `XA` and `XB` point to heap allocated arrays. Thus we do not have to reload the references to `XA` and `XB` before the loop. In the loop, we know that `fabs()` cannot modify `XL[I]`, so it does not have to be reloaded after the call to `fabs()`. This last result can be achieved with just MOD/REF analysis. Note that the `sign` macro just returns the value of `a` with the same sign as `b`.

```
CSHIFT(XA,XB,XMA,XMB,XL,BOXH,BOXL)
double XA[], XB[], XL[];
double BOXH, BOXL, XMA, XMB;
{

  int I;

  XL[0] = XMA-XMB;
  XL[1] = XMA-XB[0];
  XL[2] = XMA-XB[2];
  XL[3] = XA[0]-XMB;
  XL[4] = XA[2]-XMB;
  XL[5] = XA[0]-XB[0];
  XL[6] = XA[0]-XB[2];
  XL[7] = XA[2]-XB[0];
  XL[8] = XA[2]-XB[2];
  XL[9] = XA[1]-XB[1];
  XL[10] = XA[1]-XB[0];
  XL[11] = XA[1]-XB[2];
  XL[12] = XA[0]-XB[1];
  XL[13] = XA[2]-XB[1];

  for (I = 0; I <  14; I++) {
    if (fabs(XL[I]) > BOXH) {
      XL[I]=XL[I]-(sign(BOXL,XL[I]));
    }
  }
} /* end of subroutine CSHIFT */
```

Figure 5.30 How pointer analysis improves **water**

In another example, pointer analysis significantly improved an inner-loop in **mlink** (see Figure 5.31). In this code, we have a triply-nested loop that has a complicated address calculation and a store to that address. Unfortunately, loop indices, which are part of the address calculation, are addressed. Thus, the store in the inner-loop may modify the indices. Without pointer analysis the indices must be loaded at the beginning of the inner-loop. With pointer analysis, we are able to determine that the store does not modify the indices, and they may be kept in registers. This results in the greatly improved code seen in Figure 5.32.

5.2.3 SMALL heap model

We also examined how various heap models change the effectiveness of the analysis. The absolute results for analysis with a SMALL heap model are shown in Figures 5.33, 5.34, and 5.35. Note that in these figures we do not show the results with no analysis and MOD/REF analysis since the heap model will not affect these analyses. The ratios between the results produced by the SMALL and MEDIUM heap models without using path expressions is summarized in in Figure 5.40. Surprisingly, almost all these ratios are very close to one, which means that the more refined heap model produced little runtime improvement. In fact, in one case the ratio is less than one, which indicates that the SMALL heap model produced better runtime results than the MEDIUM heap model. However, this does not indicate that the MEDIUM heap model produced less accurate pointer analysis. More likely it indicates that the more accurate pointer analysis produced by the MEDIUM heap model caused other optimizations (perhaps the register allocator) to produce poorer code. Since the results of the analyzed code with our SMALL heap model are very close to the results with a MEDIUM heap model, and since the time and memory requirements of the SMALL heap model are always comparable or less than the requirements with a MEDIUM heap model,[9] the SMALL heap model is a good alternative to the MEDIUM heap model.

This was a surprising result to us. Of all the ways to refine a heap model, we expected going from a SMALL heap model to a MEDIUM heap model to have the most impact. We had guessed that splitting the heap by k-limiting would have less impact. The reason for this is that splitting one heap name into two names is only beneficial if we can find a memory operation that only operates on one of the two names. When we split the heap by `malloc` call site, as in the MEDIUM heap model,

[9]Some test cases could only be analyzed with a SMALL heap model. For example, **mlink** with path length four could not be analyzed with the MEDIUM heap model, but it could be analyzed with the SMALL heap model.

```
if (ch == 'i' || ch == 'I') {

  ...

  for (sys = 0; sys < FORLIM; sys++) {
    for (a = 0; a < maxall; a++) {
      for (b = 0; b < maxall; b++)
        WITH->possible[sys][a][b] = false;
    }
  }
} else {

  ...

  fscanf(speedfile, "%ld%ld%ld", &sys, &a, &b);

  ...
}
```

Figure 5.31 How pointer analysis improves **mlink**: source code

Modref Analyzed		Pointer Analyzed	
iSLDor	@readspeed_b_0 4 0 r0 => r108		
iSLDor	@readspeed_a_8 4 0 r22 => r109		
iMUL	r23 r109 => r110		
iSLDor	@readspeed_sys_16 4 0 r15 => r111		
iMUL	r24 r111 => r112		
iADD	r96 r112 => r113		
iADD	r110 r113 => r114		
iADD	r108 r114 => r115	iADD	r92 r105 => r95
bPSTor	@T93 1 0 r115 r6	bPSTor	@T333 1 0 r95 r6
iSLDor	@readspeed_b_0 4 0 r0 => r116		
iADDI	1 r116 => r117	iADDI	1 r105 => r105
iSSTor	@readspeed_b_0 4 0 r0 r117		
iCMPlt	r117 r23 => r118	iCMPlt	r105 r23 => r96
BR	L27 L32 r118	BR	L27 L32 r96

Figure 5.32 How pointer analysis improves **mlink**: machine code

Program	Path 0	Path 1	Path 2	Path 3	Path 4
tsp	651487	651487	651487	651570	651487
mlink	125723783	125722997	125722575	125722575	125722575
fft	12562160	12562142	12562142	12562142	12562142
clean	1108473	1105261	1105261	1105261	1105261
cachesim	11587501	11587501	11587501	11587501	11587501
dhrystone	552181	548186	548186	548186	548186
water	12726871	12726871	12726871	12726871	12726871
indent	867265				
allroots	1000	1000	1000	1000	1000
bc	5550822	5549718	5549239	5549239	5549239
go	486937168	486777895	486872714	486872714	486872714
bison	3331125	3327915	3327915	3327915	3327915
jpeg	36962001				
gzip(enc)	5678893	5678893	5678893	5678893	5678893
gzip(dec)	976366	976366	976366	976366	976366

Figure 5.33 SMALL heap model: total operations

Program	Path 0	Path 1	Path 2	Path 3	Path 4
tsp	51048	51048	51048	51048	51048
mlink	2419269	2419265	2419265	2419265	2419265
fft	1003608	1003599	1003599	1003599	1003599
clean	82402	80189	80189	80189	80189
cachesim	594390	594390	594390	594390	594390
dhrystone	56010	56010	56010	56010	56010
water	1069645	1069645	1069645	1069645	1069645
indent	69141				
allroots	11	11	11	11	11
bc	238488	238488	238488	238488	238488
go	20403336	20412840	20475320	20475320	20475320
bison	539606	539606	539606	539606	539606
jpeg	2503022				
gzip(enc)	198823	198823	198823	198823	198823
gzip(dec)	17233	17233	17233	17233	17233

Figure 5.34 SMALL heap model: stores

Program	Path 0	Path 1	Path 2	Path 3	Path 4
tsp	107937	107937	107937	107937	107937
mlink	26515862	26515294	26515183	26515183	26515183
fft	1211787	1211787	1211787	1211787	1211787
clean	180429	179091	179091	179091	179091
cachesim	1900619	1900619	1900619	1900619	1900619
dhrystone	50021	46021	46021	46021	46021
water	1673879	1673879	1673879	1673879	1673879
indent	147735				
allroots	141	141	141	141	141
bc	812013	810909	810430	810430	810430
go	83945430	83851902	83865727	83865727	83865727
bison	552589	551519	551519	551519	551519
jpeg	6594787				
gzip(enc)	835906	835906	835906	835906	835906
gzip(dec)	135636	135636	135636	135636	135636

Figure 5.35 SMALL heap model: loads

we can easily imagine memory operations that operate on the memory from only one call site or some subset of the `malloc` call sites. However, if we split by `k`-limiting, it is more difficult to imagine a memory operation operating on only a subset of the k names (*e.g.*, it is difficult to find code with a memory operation that only operates on, say, the second element of a linked-list). Since the heap model that we expected to have the most impact had very little impact, we suspect that the other ways of refining the heap will produce little benefit also. However, there is one heap model refinement that was not tested that may have a substantial impact – splitting the heap based on the field names of the structure being allocated. We think this may be beneficial because we can imagine memory operations that operate on only one field of a heap-allocated structure. Admittedly, we also imagined that a MEDIUM heap model would produce a significant improvement over a SMALL heap model, so we can only cautiously recommend this as an area for research.

We examined how analysis with a MEDIUM heap model can improve the performance of analyzed code over analysis with a SMALL heap model. A MEDIUM heap model was useful for improving the performance of **mlink**, the program in which such a model made the most difference. Most of the improvement came in the function `segup`. Over half the loads removed by changing heap models came from this function. There were many places in which our MEDIUM heap model allowed changes to the code. One place we found is seen in Figure 5.36. This example may or may not be the reason for most of the improvement due to this function. In this example we have a triply nested loop. Within this loop we have a load, `*LINK->p`, from some heap allocated structures. We also have a store to a heap allocated structure `segval[i]`. With a MEDIUM heap model we can determine that these two memory accesses do not interfere, thus we can move the load outside the outermost loop. With a SMALL heap model we cannot determine that these two accesses do not interfere. Thus, we cannot move the load outside any of the loops. We were surprised that this sort of pattern did not occur more often in our test suite.

5.2.4 LARGE heap model

Another heap model that was used was splitting the heap by user defined heap allocators. Five programs were found to have their own heap allocators: **clean**, **indent**, **bc**, **bison**, and **gzip**.

The absolute results for this analysis are shown in Figures 5.37, 5.38, and 5.39.[10] The ratios between the results produced by the LARGE and baseline heap models without

[10]Note that the results for **clean** were generated using a newer version of our optimizer than the other performance results for analyzed code in this thesis. This was necessary because a bug was discovered late in the writing of this thesis that affected the analysis of **clean** with a LARGE heap

```
for (nonzindex = 0; nonzindex < nonzcount; nonzindex++) {
  ...
  for(i = 0; i < step1; i++) {
    ...
    for(j = 0; j < step2; j++) {
      ...
      if ((!sexdif) || ((*LINK->p)->male))
      ...
      segval[i] += newwith3[second+j] * val;
      ...
    }
  }
}
```

Figure 5.36 Splitting the heap helps analysis

Program	Path 0	Path 1
clean	1017021	1017021
indent	865622	
bc	5548284	5536821
bison	3326755	3326473
gzip(enc)	5678893	5678893
gzip(dec)	976366	976366

Figure 5.37 LARGE heap model: total operations

Program	Path 0	Path 1
clean	55970	55970
indent	69141	
bc	238486	238486
bison	539597	539597
gzip(enc)	198823	198823
gzip(dec)	17233	17233

Figure 5.38 LARGE heap model: stores

Program	Path 0	Path 1
clean	146949	146949
indent	147159	
bc	809476	802617
bison	550720	550626
gzip(enc)	835906	835906
gzip(dec)	135636	135636

Figure 5.39 LARGE heap model: loads

using path expressions is summarized in in Figure 5.40. These results show that more accurate heap models do not significantly improve the performance of the analyzed code.

5.2.5 Path expressions

We also examined how path expressions improve analysis and cause code to run faster. In our fourteen test cases, path expressions had the greatest impact in **dhrystone**. All of the improvement occurred in the function `main`. There were two situations within `main` that allowed path expressions to improve the code.

In the first example (see Figure 5.2.5), `Enum_Loc` will unnecessarily be loaded from memory in the `if` statement if path expressions are not used. `Enum_Loc` is already available from the assignment at the beginning of the code. Unfortunately, without path expressions, our pointer analysis will show that the call to `Proc_1` may modify `Enum_Loc`, and thus we must reload `Enum_Loc` at the `if` statement. With path expressions, this load is unnecessary since we can determine that this call to `Proc_1` does not modify `Enum_Loc`. Path expressions allow us to determine this by allowing us to tailor the MOD/REF effect for each call site to a function. Within `Proc_1` there is a call to `Proc_6`. `Proc_6` modifies the variable whose address is given as its second parameter. The call within `Proc_1` does not pass the address `Enum_Loc` as its second parameter, but another call to `Proc_6` does. Without path expressions we must combine the MOD/REF effect for all the calls to `Proc_6` and put this effect at all call sites. Thus the call to `Proc_6` in `Proc_1` and `Proc_1` itself must be conservatively and unnecessarily assumed to modify `Enum_Loc`. With path expressions, it can be determined that the call to `Proc_6` in `Proc_1` does not modify `Enum_Loc`, even though another call to `Proc_6` does modify `Enum_Loc`.

	SMALL	MEDIUM	LARGE	Range
Operations	100.1	100.0	99.9	99.8-101.0
Stores	100.2	100.0	100.0	100.0-102.9
Loads	100.3	100.0	99.8	99.6-103.9

Figure 5.40 Summary of memory model impact on code performance

model. We could not regenerate the old version of the optimizer. Thus, the ratios for **clean** with the LARGE heap model versus our baseline model show the ratios when using the new optimizer. In particular, we did not use the performance results for our baseline model in Figures 5.24, 5.26, and 5.28 since these results were produced with our old optimizer.

A similar thing occurs in our second example (see Figure 5.2.5). In this case the variable Int_3_Loc is unnecessarily reloaded on the last statement. It is already available as a parameter to the call to Proc_8. As in the previous example, without path expressions, we must unnecessarily assume that the call to Proc_1 may modify Int_3_Loc. This is due to a call to Proc_7 in Proc_1. Proc_7 modifies the variable whose address is passed as its third parameter. The call to Proc_7 within Proc_1 does not pass the address of Int_3_Loc as its third parameter, but another call to Proc_7 does. This causes inaccuracy when the MOD/REF effect from all the call sites to Proc_7 are combined. This inaccuracy can be removed when path expressions are used.

Both of these examples occur in a loop that is iterated a user specified number of times (in our case this is 2000). Path expressions will remove one load in each example. Thus, these two examples explain why path expressions will remove 4000 loads in the execution of **dhrystone**. This accounts for all the loads removed by path expressions in **dhrystone**.

5.2.6 Cache results

We also ran our analyzed programs in two models that contained a cache simulator. In the first model we simulated a single-level 2KB, 2-way set-associative write-back cache with a line size of 32 bytes. In the second model, we simulated a two-level cache with a 1KB direct-mapped, write-through L1 cache, and a 64KB 4-way set-associative, write-back L2 cache, each with a line size of 16 bytes. In order to give a background to interpret our cache results, Figure 5.43 shows the impact of varying the level of analysis on the number of cache accesses. These numbers are normalized with the version having no analysis set to 100.0. The "% of total" line shows the percentage of operations in the unanalyzed program that accessed the cache. The other figures in this section follow an identical format, except that instead of the "number of cache

```
Enum_Loc = Ident_2;
...
Proc_1 (Ptr_Glob);
...
  if (Enum_Loc == Func_1 (Ch_Index, 'C'))
```

Figure 5.41 Path expressions help: example 1

```
Proc_8 (Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
...
Proc_1 (Ptr_Glob);
...
Int_1_Loc = Int_2_Loc / Int_3_Loc;
```

Figure 5.42 Path expressions help: example 2

accesses," we present the number of cache misses for a particular cache level. When interpreting the importance of these results, the "% of total" column and the cost of a cache access and a cache miss are very important. Overall, the number of cache accesses is the most important statistic since most memory requests are serviced at the highest cache level. However, this emphasis should be balanced by the greater cost of acccessing lower cache levels.

Figure 5.44 shows the results for the single-level cache model. This table is sorted so that programs that have a higher percentage of operations that produced a cache miss are placed on the top of the figure. Comparisons of the number of misses for **tsp** and **dhrystone** are not meaningful since the total number of misses is very small. This figure shows that MOD/REF analysis and pointer analysis usually decreased the number of cache misses for programs for which the number of cache misses was significant (15 out of 24 cases). However, there were important exceptions. For **go**, **bc** and the pointer analyzed version of **indent**, the percentage of misses was significantly increased. This was especially bad for **go** since such a large percentage of its operations caused cache misses. On the other hand, both MOD/REF analysis and pointer analysis reduced the number of misses for **bison** by over half. Pointer analysis usually produced better results than MOD/REF analysis for programs in the top half of the figure while MOD/REF analysis produced better results for programs in the lower half. For **indent**, MOD/REF analysis generated much better results than pointer analysis. We were not able to determine why this is the case, but this result is not unexplainable. One possiblity is that pointer analysis may produce more register pressure which causes a register spill. The register might not be serviced by the cache causing more cache misses.

Figure 5.45 shows the results for the L1 cache of two-level cache. For this level, analysis occasionally increased the number of misses by a small percentage. However, for most of the programs in our test suite, the number of misses was reduced by large

Program	% of total	Original	MOD/REF	Path 0
tsp	29.81	100.00	75.72	73.06
mlink	24.25	100.00	82.78	77.66
fft	22.34	100.00	75.09	72.42
clean	25.27	100.00	76.02	75.00
dhrystone	21.26	100.00	95.30	82.81
water	27.90	100.00	83.66	65.01
indent	29.70	100.00	75.06	75.14
allroots	16.75	100.00	89.66	87.36
bc	19.51	100.00	96.78	96.24
go	21.94	100.00	95.87	91.00
bison	38.52	100.00	73.75	73.67
jpeg	24.48	100.00	100.67	100.66
gzip(enc)	19.54	100.00	94.49	91.62
gzip(dec)	15.76	100.00	98.25	97.94

Figure 5.43 Normalized cache accesses

percentages. Most notably, **bison** and **tsp** had the number of misses reduced by multiple factors. For these programs this should be especially benefical since misses occur for a large percentage of the total operations. At this level pointer analysis outperformed MOD/REF analysis in most cases. The results for the L2 cache of this model are shown in Figure 5.46. Only **go**, **fft**, and **mlink** had over 10,000 misses. For the other programs, the effect of the analyses never changed the number of misses by more than a few hundred. This indicates that the memory required for these programs was usually less than the 64KB size of the L2 cache. For the programs that could not fit in the L2 cache, our analyses had a significant impact on only two cases, **go** and **mlink**. For **mlink**, there was a minor improvement, but for **go**, MOD/REF analysis nearly doubled the number of misses.

Overall the impact of pointer analysis and MOD/REF analysis on memory performance was good. Analysis usually decreased the number of memory references in a program. Our cache performance statistics show that at the highest cache level, this reduction in the number of memory references usually also reduced the number of cache misses. For most of these programs, such as **water**, analysis increased the miss rate,[11] but this was only because analysis was able to remove a greater percentage of memory accesses that produced cache hits than cache misses. For other programs, such as **bison**, analysis was able to reduce the number of accesses and reduce the miss percentage. There were programs, such as **go**, where the number of cache misses increased. However, this was compensated by the fact that the number of accesses was reduced. Although there were borderline cases, MOD/REF analysis and pointer analysis usually reduced the memory traffic and cache misses. These effects will result in reduced execution times for our test programs.

[11]We can determine this because the normalized number of accesses is less than the normalized number of misses. This indicates that a greater fraction of accesses than misses was removed, thus increasing the miss rate.

Program	% of total	Original	MOD/REF	Path 0
go	7.13	100.00	115.57	109.54
gzip(enc)	5.87	100.00	102.42	99.24
clean	4.66	100.00	93.17	89.43
mlink	4.16	100.00	92.65	95.30
jpeg	4.06	100.00	99.65	99.37
fft	2.15	100.00	95.57	92.13
indent	1.66	100.00	67.94	124.11
allroots	1.15	100.00	100.00	91.67
bison	1.01	100.00	46.23	47.15
water	0.98	100.00	90.37	97.94
gzip(dec)	0.87	100.00	101.28	103.82
bc	0.79	100.00	137.60	137.83
tsp	0.09	100.00	470.38	319.36
dhrystone	0.00	100.00	121.43	121.43

Figure 5.44 Single-level cache model: normalized misses

Program	% of total	Original	MOD/REF	Path 0
go	8.60	100.00	107.42	103.18
bison	8.45	100.00	8.65	7.97
mlink	7.22	100.00	88.22	93.41
gzip(enc)	6.56	100.00	103.93	100.52
clean	5.98	100.00	90.93	88.76
tsp	5.47	100.00	30.47	30.54
jpeg	4.73	100.00	100.32	98.49
water	3.38	100.00	89.07	87.04
indent	3.31	100.00	77.85	83.23
fft	2.54	100.00	89.30	77.48
allroots	2.21	100.00	69.57	65.22
bc	2.14	100.00	91.34	100.25
gzip(dec)	1.51	100.00	100.54	105.17
dhrystone	0.67	100.00	199.45	199.45

Figure 5.45 Two-level cache model: normalized misses (level one)

Program	% of total	Original	MOD/REF	Path 0
allroots	1.44	100.00	106.67	100.00
fft	0.84	100.00	99.58	99.68
gzip(enc)	0.48	100.00	98.94	98.66
gzip(dec)	0.33	100.00	101.13	100.67
clean	0.26	100.00	99.86	100.71
bison	0.25	100.00	100.33	100.67
go	0.21	100.00	188.85	109.05
tsp	0.18	100.00	99.16	99.24
mlink	0.17	100.00	88.83	98.55
indent	0.13	100.00	98.08	106.98
bc	0.03	100.00	103.45	104.47
water	0.02	100.00	97.59	97.16
jpeg	0.01	100.00	100.53	105.51
dhrystone	0.00	100.00	95.00	95.00

Figure 5.46 Two-level cache model: normalized misses (level two)

5.3 Optimizer performance

We optimized our `ILOC` files by using a PERL script to control the optimization process. We also used the script to record the time needed to optimize all the files in each program. These results are shown in Figure 5.47. For comparison, in the far right column, we also show the time needed to analyze the program with no path expressions with the MEDIUM heap model. In order to see the impact of the analyses on optimization time more clearly, we standardized the results so that the optimization time of the unanalyzed program was 1.000 (see Figure 5.48). This shows that MOD/REF analysis usually increases the amount of time needed to optimize a program while pointer analysis has the opposite effect. Pointer analysis with no path expressions speeds up optimization enough so that the time required for analysis and optimization (Analysis Time column plus Path 0 column) is usually less than the time required for optimization with no analysis (Original column). However, there were three exceptions, (**clean**, **bc**, and **jpeg**), which accounted for 22% of our test cases. Also, for the **jpeg** case, this increase was huge (*i.e.*, the cost of analysis was nearly forty times greater than the cost of optimization with no pointer analysis.

Program	Original	Modref	Path Length					Analysis Time
			0	1	2	3	4	
tsp	27.0	30.2	23.3	23.6	23.8	22.6	25.2	0.2
mlink	568.7	576.6	361.9	371.0	334.9	351.9		32.9
fft	37.8	25.7	23.9	27.3	22.3	23.8	25.5	0.2
clean	243.2	346.2	318.0	336.0	312.1	306.1	348.6	21.8
cachesim	96.5	126.2	91.0	87.3	79.4	81.9	88.7	3.7
dhrystone	15.7	14.7	14.8	12.1	11.9	13.1	12.7	0.1
water	70.2	44.7	43.1	38.9	45.5	38.5	45.0	0.4
indent	298.0	299.4	211.6					42.9
allroots	9.1	8.4	8.5	8.1	7.8	7.1	8.9	0.0
bc	210.6	222.8	212.2	218.4	208.4	204.7		30.6
go	904.9	1088.5	839.8	817.8	635.6	919.0	919.0	46.6
bison	368.0	389.7	340.9	347.2	316.4	304.0	368.6	8.6
jpeg	292.1	413.2	675.1					11539.5
gzip	196.0	223.4	184.5	183.9	175.1	173.0	205.9	5.9

Figure 5.47 Optimization times (s): absolute

Program	Original	Modref	Path Length					Analysis Time
			0	1	2	3	4	
tsp	1.000	1.119	0.866	0.874	0.881	0.838	0.933	0.009
mlink	1.000	1.014	0.636	0.652	0.589	0.619		0.058
fft	1.000	0.680	0.632	0.723	0.589	0.629	0.675	0.006
clean	1.000	1.424	1.308	1.381	1.283	1.258	1.433	0.090
cachesim	1.000	1.308	0.943	0.904	0.823	0.849	0.919	0.038
dhrystone	1.000	0.936	0.944	0.767	0.760	0.835	0.810	0.008
water	1.000	0.637	0.614	0.553	0.648	0.549	0.640	0.006
indent	1.000	1.005	0.710					0.144
allroots	1.000	0.921	0.935	0.887	0.863	0.778	0.982	0.004
bc	1.000	1.058	1.008	1.037	0.990	0.972		0.145
go	1.000	1.203	0.928	0.904	0.702	1.016	1.016	0.052
bison	1.000	1.059	0.926	0.943	0.860	0.826	1.001	0.023
jpeg	1.000	1.415	2.312					39.511
gzip	1.000	1.140	0.941	0.938	0.893	0.883	1.051	0.030

Figure 5.48 Optimization times: standardized (1.000 = original)

Chapter 6

Register Promotion

We have also developed a technique, register promotion, to utilize the information generated by pointer analysis. Register promotion improves code by allowing a value that normally resides in memory to reside in a register for some portions of the code. This is done by identifying sections of the code in which it is safe to place the value in a register. Before entering such a section, the value is "promoted" (*i.e.*, loaded) from its memory location to a register. Within the section, references to this value are rewritten to refer to the register. Upon exit from the section, the value is "demoted" (*i.e.*, stored) to a memory location.

The compiler performs promotion in the early phases of optimization. It rewrites the `ILOC` to keep additional values in a register. However, subsequent actions by the register allocator can "undo" a promotion. At the time that promotion occurs, the compiler cannot accurately predict the availability of a register to hold the promoted value. If the register allocator discovers that demand for registers exceeds supply, it must "spill" some values back to memory. The promoted values compete for registers on an equal footing with other values; and as a result, some of them may get spilled.

6.1 The algorithm

The algorithm that we have developed is relatively simple. It proceeds as follows:

1. *interprocedural analysis*—The compiler performs an interprocedural analysis to disambiguate memory references. The results are used to shrink the tag sets for references and procedure calls.

2. *gather initial information*—For each block b, the compiler computes two sets. $\texttt{B\_EXPLICIT}_b$ contains all tags referenced by an explicit memory operation in b. $\texttt{B\_AMBIGUOUS}_b$ contains all tags referenced ambiguously in b, through procedure calls or pointer-based memory operations where the pointer contains multiple tags.

3. *find loop structure*—The compiler computes dominator information to identify loop nests using an algorithm due to Lengauer and Tarjan [27].

4. *analyze loop nests*—For each loop l, the compiler solves the equations shown in Figure 6.1. The set $\texttt{L\_PROMOTABLE}_l$ contains the tags that may safely be promoted inside loop l.

5. *rewrite the code*—For each tag that is in some $\texttt{L\_PROMOTABLE}_l$, a virtual register v is created. All references to the tag in loops for which the tag is promotable are converted to a copy involving v.[12]

6. *promote the tags*—A tag that has had its accesses rewritten to use a virtual register must be loaded into its virtual register before entering the outermost loop in which it is promotable. It also must be stored to at the loop exits. The set of tags that needs to be loaded and stored around a loop l is in $\texttt{L\_LIFT}_l$.

The equations from Figure 6.1 merit some additional explanation. $\texttt{B\_EXPLICIT}_b$ and $\texttt{B\_AMBIGUOUS}_b$ are computed in a simple linear pass over each block. The pass must examine each operation and its tag set. Equations (1) and (2) simply aggregate together the information for all the blocks in a loop. Equation (3) is solved once per loop; it computes the set of values that are only referenced explicitly in the loop. If a tag `t` is in $\texttt{L\_PROMOTABLE}_l$ for loop l, the loop can be rewritten safely to keep the value associated with `t` in a register. Finally, equation (4) ensures that a tag `t` is only loaded and stored around the outermost loop where it may be promoted.

What have we accomplished? As presented, the algorithm promotes references to a scalar variable in a loop if all the references to the scalar variable in the loop

$$\texttt{L\_EXPLICIT}_l = \bigcup_{b \in l} \texttt{B\_EXPLICIT}_b \tag{6.1}$$

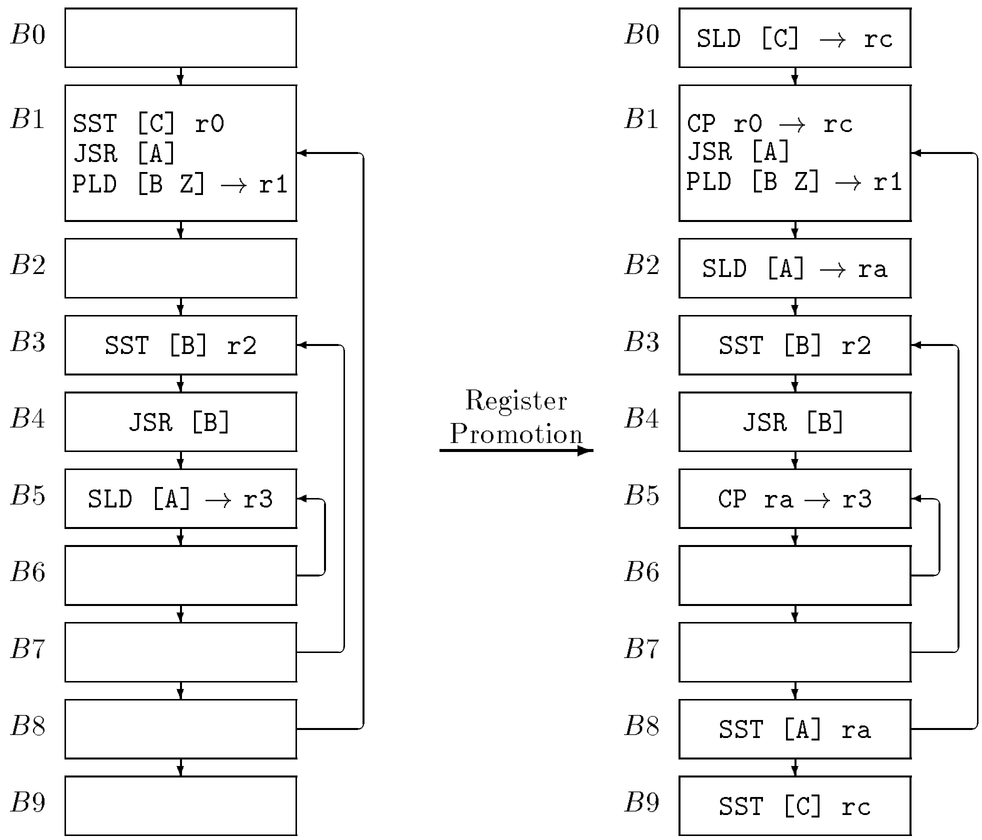
$$\texttt{L\_AMBIGUOUS}_l = \bigcup_{b \in l} \texttt{B\_AMBIGUOUS}_b \tag{6.2}$$

$$\texttt{L\_PROMOTABLE}_l = \texttt{L\_EXPLICIT}_l - \texttt{L\_AMBIGUOUS}_l \tag{6.3}$$

$$\texttt{L\_LIFT}_l = \begin{cases} \texttt{L\_PROMOTABLE}_l & \text{if } l \text{ is outermost loop} \\ \texttt{L\_PROMOTABLE}_l - \texttt{L\_PROMOTABLE}_{surrounding\_loop(l)} & \text{otherwise} \end{cases} \tag{6.4}$$

Figure 6.1 Equations for register promotion

[12]The copies are subject to coalescing by the register allocator [4]. It is quite effective at eliminating copies like these.



Loop Nest and Relevant Code

Block Information										
Set	0	B1	B2	B3	B4	B5	B6	B7	B8	B9
B_EXPLICIT		C		B		A				
B_AMBIGUOUS		A B Z			B					

Loop Information						
Loop	Landing Pad	Exit	L_EXPLICIT	L_AMBIGUOUS	L_PROMOTABLE	L_LIFT
B1-B8	B0	B9	A B C	A B Z	C	C
B3-B7	B2	B8	A B	B	A	A
B5-B6	B4	B7	A		A	

Figure 6.2 An example

are explicit. It does not promote references based on pointers that may point to multiple objects; neither does it promote array references. The promoted variables are scalars that the compiler did not enregister because it lacked the information to show that enregistering them was safe. Section 6.3 discusses one technique for extending the domain of promotion to include some array and more pointer-based values. The algorithm only examines references inside loops; our implementation of partial redundancy elimination[30] uses memory tag information to achieve most of the effects of promotion in straight-line code.

What does this algorithm cost? The cost of the interprocedural analysis used to support register promotion varies with both the algorithm used and the desired precision of the information. The promotion algorithm itself runs efficiently. Its complexity is expressed as a function of the following variables that characterize a program.

C	code size
T	number of tags
L	number of loops
X	maximum number of exits in a loop
B	number of basic blocks
E	number of edges in CFG

Computing `B_EXPLICIT` and `B_AMBIGUOUS` takes a simple pass over the code. In each block, it examines each statement and, possibly, each tag set. This requires $\mathbf{O}(CT)$ time, worst case. The dominator algorithm used to find the loop structure can be implemented to require $\mathbf{O}(E\alpha(E,B))$ time, where $\alpha(E,B)$ is related to a functional inverse of Ackermann's function [27]. Computing `L_EXPLICIT` and `L_AMBIGUOUS` requires $\mathbf{O}(LBT)$ time, while `L_PROMOTABLE` and `L_LIFT` require $\mathbf{O}(LT)$ time. Rewriting the code requires $\mathbf{O}(C)$ time to convert memory operations to copies, plus $\mathbf{O}(TLX)$ time to insert loads and stores at loop landing pads and loop exits. Thus, the overall time bound is

$$\mathbf{O}(CT + E\alpha(E,B) + LBT + LT + C + TLX),$$

which simplifies to

$$\mathbf{O}(E\alpha(E,B) + T \cdot (C + LB + LX)).$$

In practice, it runs quickly.

6.2 An example

To make this discussion more concrete, consider the example shown in Figure 6.2. It shows a triply nested loop, along with some of the code that populates the loop. (The remaining code is assumed to have no impact on the example.) The instructions

```
for (i=0; i<DIM_X; i++) {
  B[i]=0;
  for (j=0; j<DIM_Y; j++) {
    B[i]+=A[i][j];
  }
}
```

Original code

```
for (i=0; i<DIM_X; i++) {
  rb=0;
  for (j=0; j<DIM_Y; j++) {
    rb+=A[i][j];
  }
  B[i]=rb;
}
```

Transformed code

Figure 6.3 Promoting array references

are presented in an abstracted form; each shows its tag list followed by any relevant registers. The mnemonics have simple meanings.

SST	Scalar store
SLD	Scalar load
CP	Register copy
PST	Pointer-based store
PLD	Pointer-based load
JSR	Jump to subroutine

The version on the left shows the code before register promotion. The version on the right shows the results of register promotion. Notice that each loop has an explicit landing pad before its header and an explicit exit block. Our compiler automatically inserts landing pads and exits as part of constructing the control-flow graph; empty blocks are automatically removed after optimization.

The tables at the bottom of the figure show the local information computed for the example, `B_EXPLICIT` and `B_AMBIGUOUS`, as well as the sets computed for the loops. The `L_PROMOTABLE` and `L_LIFT` sets concisely summarize the situation. The value associated with tag A is promotable in the two inner loops but not the outer loop. The `JSR` instruction in block B1 references A ambiguously, so it cannot be promoted in that loop. The value associated with tag B is referenced ambiguously in loop B3-B7; since it is not referenced in loop B5-B6, no opportunity for promoting it exists. Finally, the value associated with tag C is never referenced ambiguously. Since it is referenced in the outer loop (B1-B8), it is promotable in that loop. The `L_LIFT` set correctly shows that A should be promoted in loop B3-B7 rather than loop B5-B6 since loop B3-B7 contains loop B5-B6.

When the compiler rewrites the code, it promotes C in loop B1-B8 and A in loop B3-B7. Thus, it inserts a scalar load of C into `rc` in loop B1-B8's landing pad (block B0) and a scalar store into loop B1-B8's exit block (B9). The store in block B1 becomes a copy into `rc`. To promote A, it inserts a scalar load of A into `ra` in loop B3-B7's landing pad (B2), and a scalar store into loop B3-B7's exit block (B8). The load in B5 becomes a copy out of `ra`. The other instructions remain unchanged.

The net result is to replace the scalar load in the innermost loop with a copy operation and a load/store pair two loops farther out. The scalar store in the outer loop is replaced with a copy operation and a load/store pair outside that loop. In many cases, the register allocator can coalesce away these copies.

6.3 Handling pointer-based references

The algorithm from Section 6.1 only promotes scalar variables that are explicitly referenced. Pointer-based loads and stores that may point to multiple locations cannot be modified. Consider, for example, the code shown on the left hand side of Figure 6.3. The inner loop uses a single value of `B[i]` per iteration of the outer loop; since `i` does not change, the address of `B[i]` is invariant. Thus, the compiler should rewrite the code as shown on the right by promoting `B[i]` into a register `rb`. This eliminates a load before the reference to `B[i]` in the inner loop and a store after it. To achieve this, however, the compiler must recognize that `B[i]` refers to the same location in each iteration of the inner-loop and that only one way to reference the code is possible in the inner loop. The analysis described earlier cannot do that.

We developed another algorithm to promote some pointer-based references to multiple locations. In particular, it finds memory references, r, where the base register, b, is invariant in a loop and the only accesses in the loop to the tags accessed by r are through the invariant base register b. This algorithm relies on loop-invariant code motion to identify the loop-invariant base registers and place the computation of these registers outside a loop. When it finds memory references satisfying these conditions, it promotes the reference into a register using the same rewriting scheme as before—a load before each loop entry, a store at each loop exit, and a copy at each reference. These conditions include the example from Figure 6.3.

Anecdotally, the pointer-based promotion scheme is a success. When its conditions apply, it produces the code that might be expected of a good assembly programmer. For example, it produces a loop equivalent to the transformed code shown in Figure 6.3, after coalescing removes the copy operations. In our suite of test programs, however, the measured improvements were not overwhelming when compared with scalar promotion. For total operations executed, pointer-based promotion hurt performance for one program and had no effect on nine others. The improvements in three of the other four programs were less than 1% of the improvement due to scalar promotion. In **fft**, the only significant success, pointer-based promotion was able to remove 48.3% more operations, 48.3% more stores, and 48.4% more loads than scalar promotion was able to remove. This accounted for 0.41% of the stores and 0.34% of the loads in the execution of **fft**. The reason for this disappointing performance may be that the restrictions are too strict; it may be that the promotable pointer-based references in our collection of programs are relatively unimportant to performance. We intend to continue to investigate this set of problems.

6.4 Planned improvements

Our current register promoter misses some opportunities. We are interested in extending this work to increase its coverage of real programs.

- The loop-based approach to analysis and transformation causes the promoter to overlook situations that occur outside loop nests. There should be many cases where it is profitable to promote values to registers in straight-line code.

  In our compiler, partial redundancy elimination handles many of these cases in straight-line code. It uses the tag fields to eliminate redundant loads. It must treat stores more conservatively. Extending the promoter could improve the behavior for these stores.

- The scheme described in Section 6.3 handles a set of relatively simple array references. Some of the more complex examples require detailed dependence analysis or an equivalent technique to reason about conflicts with other references to the same array inside the loop [17, 28]. For example, Carr used dependence analysis to detect consistent patterns of cross-iteration reuse in Fortran and to promote the corresponding values into scalar temporaries that ended up in registers [8].

  We are interested in expanding the set of array references promoted by the compiler. Our work to date has focussed on poor code that results from lack of information about the behavior of other procedures and pointer-based memory operations. As we delve deeper into array promotion, we will need to improve our analysis of subscripts.

As in any experimental study, examining the code that comes out of the compiler suggests additional areas of improvement. Further cases for improvement will suggest themselves as we continue this work.

However, we must sound a note of caution. Register promotion increases the demand for registers—often called *register pressure*. As we improve the promoter, we increase its ability to generate an intermediate code program that requires spilling in the register allocator. Carr discovered this effect in his work on scalar replacement in Fortran [9]; beyond some point, the memory accesses removed by the transformation were balanced by the spills added during register allocation. He adopted a bin-packing discipline to "throttle" the promotion process. As we extend our work, we will undoubtedly encounter the same problem and need a similar solution to moderate register pressure.

6.5 Results

Register promotion had a small effect on the total number of operations executed (see Figures 6.4, 6.5, and 6.6). The main benefit of register promotion is the removal of memory operations–stores and loads (see Figures 6.7, 6.8, 6.9, 6.10, 6.11, and 6.12). In several of the applications, promotion removed a large fraction of the stores and many of the loads. In other applications, it found few, if any, opportunities. When it found opportunities, the promoter often made significant improvements. If memory operations take more cycles than other operations, as in many modern machines, the positive impact of promotion will be greater.

In some cases, the net effect of promotion was a minor performance degradation. This is not surprising when we look at the total operations executed (see Figures 6.4 and 6.5). Register promotion replaces some memory operations with copies and adds memory operations to promote variables. Thus, by itself, register promotion will always increase the total number of operations executed. In combination with other optimization passes, register promotion may decrease the total operations executed by exposing some optimization opportunities. Performance degradation in terms of memory operations executed was caused by two effects, promoting rarely used or conditionally used values and increasing register pressure. For example, in **bison**, values were promoted that were only accessed on an error condition. In **water**, register promotion was able to promote twenty-eight values for one loop nest. Unfortunately, this caused the register allocator to spill values, which resulted in a performance loss compared to no register promotion.[13] Most of the improvements were the result of global variables that are normally placed in memory being promoted to registers.

The results also show that the improved information derived from pointer analysis does not greatly improve the results of register promotion. This does not warrant a conclusion that pointer analysis is unprofitable; it does suggest that MOD/REF

[13] It might be expected that the allocator would simply spill some subset of the twenty-eight promoted values and avoid the actual performance degradation. Our compiler uses a graph-coloring allocator [4]. These allocators are known to "over-spill" in tight situations.

Program	*analysis*	*without*	*with*	*difference*	*% removed*
tsp	modref	657271	657271	0	0.00
	pointer	651487	651487	0	0.00
	path 1	651487	651487	0	0.00
	path 2	651487	651487	0	0.00
	path 3	651487	651487	0	0.00
	path 4	651487	651487	0	0.00
mlink	modref	126763483	126772024	-8541	-0.01
	pointer	124578327	124425317	153010	0.12
	path 1	124577258	124424420	152838	0.12
	path 2	124576824	124423998	152826	0.12
	path 3	124576824	124423998	152826	0.12
fft	modref	12636446	12635914	532	0.00
	pointer	12575749	12550636	25113	0.20
	path 1	12575740	12550618	25122	0.20
	path 2	12575740	12550618	25122	0.20
	path 3	12575740	12550618	25122	0.20
	path 4	12575740	12550618	25122	0.20
clean	modref	1113150	1099823	13327	1.20
	pointer	1117631	1108473	9158	0.82
	path 1	1119738	1105261	14477	1.29
	path 2	1119738	1105261	14477	1.29
	path 3	1119738	1105261	14477	1.29
	path 4	1119738	1105261	14477	1.29
cachesim	modref	11588600	11589109	-509	-0.00
	pointer	11586991	11587501	-510	-0.00
	path 1	11586991	11587501	-510	-0.00
	path 2	11586991	11587501	-510	-0.00
	path 3	11586991	11587501	-510	-0.00
	path 4	11586991	11587501	-510	-0.00
dhrystone	modref	582185	582185	0	0.00
	pointer	552181	552181	0	0.00
	path 1	548186	548186	0	0.00
	path 2	548186	548186	0	0.00
	path 3	548186	548186	0	0.00
	path 4	548186	548186	0	0.00

Figure 6.4 Register promotion: total operations

Program	*analysis*	*without*	*with*	*difference*	*% removed*
water	modref	13478252	13476586	1666	0.01
	pointer	12725764	12726871	-1107	-0.01
	path 1	12725764	12726871	-1107	-0.01
	path 2	12725764	12726871	-1107	-0.01
	path 3	12725764	12726871	-1107	-0.01
	path 4	12725764	12726871	-1107	-0.01
indent	modref	873264	867212	6052	0.69
	pointer	873249	867180	6069	0.69
allroots	modref	1010	1010	0	0.00
	pointer	1000	1000	0	0.00
	path 1	1000	1000	0	0.00
	path 2	1000	1000	0	0.00
	path 3	1000	1000	0	0.00
	path 4	1000	1000	0	0.00
mybc	modref	5574493	5561348	13145	0.24
	pointer	5563470	5550821	12649	0.23
	path 1	5558913	5546264	12649	0.23
	path 2	5559372	5546723	12649	0.23
	path 3	5559372	5546723	12649	0.23
go	modref	489094926	496832956	-7738030	-1.58
	pointer	479136877	486937168	-7800291	-1.63
	path 1	478978573	486777895	-7799322	-1.63
	path 2	479073392	486872714	-7799322	-1.63
	path 3	479073392	486872714	-7799322	-1.63
	path 4	479073392	486872714	-7799322	-1.63
bison	modref	3333123	3334118	-995	-0.03
	pointer	3330130	3331125	-995	-0.03
	path 1	3326926	3327915	-989	-0.03
	path 2	3326926	3327915	-989	-0.03
	path 3	3326926	3327915	-989	-0.03
	path 4	3326926	3327915	-989	-0.03
jpeg	modref	36963117	36963318	-201	-0.00
	pointer	36961800	36962001	-201	-0.00

Figure 6.5 Register promotion: total operations (cont.)

Program	*analysis*	*without*	*with*	*difference*	*% removed*
gzip(enc)	modref	5814665	5711089	103576	1.78
	pointer	5805907	5678893	127014	2.19
	path 1	5805907	5678893	127014	2.19
	path 2	5805907	5678893	127014	2.19
	path 3	5805907	5678893	127014	2.19
	path 4	5805907	5678893	127014	2.19
gzip(dec)	modref	977247	976879	368	0.04
	pointer	976879	976366	513	0.05
	path 1	976879	976366	513	0.05
	path 2	976879	976366	513	0.05
	path 3	976879	976366	513	0.05
	path 4	976879	976366	513	0.05

Figure 6.6 Register promotion: total operations (cont.)

analysis is a good basis for evaluating the benefits of improved analysis. An example where pointer analysis was required to promote a value arose in **fft**.

```
for (I = begin; I < end; I++)
  for (J = 0; J < N3; J++)
    for (K = 0; K < N1; K++)
      {
        index3 = (I*N3+J)*N1+K;
        index1 = (I*N3+J)*N1*2+K;
        T1 = pow(X3[index3], (double) KT) ;
        X2[index1] = T1 * X1[index1];
        X2[index1+N1] = T1 * X1[index1+N1];
      }
```

`T1`'s address is taken elsewhere in this code. `X2` is a pointer, so the stores through it may modify `T1`. Thus `T1` is not promotable with just MOD/REF analysis. Pointer analysis can discover that the stores through `X2` cannot modify `T1`, and thus `T1` can be promoted.

Finally, some of the improvement due to register promotion was hidden because other passes in the optimizer achieve similar results. For example, loop invariant code motion can remove a load of a constant value out of a loop. Register promotion's main benefit seems to be transforming multiple stores of a promoted variable in a loop

Program	*analysis*	*without*	*with*	*difference*	*% removed*
tsp	modref	51048	51048	0	0.00
	pointer	51048	51048	0	0.00
	path 1	51048	51048	0	0.00
	path 2	51048	51048	0	0.00
	path 3	51048	51048	0	0.00
	path 4	51048	51048	0	0.00
mlink	modref	2509189	2505951	3238	0.13
	pointer	2442134	2352204	89930	3.68
	path 1	2442131	2352200	89931	3.68
	path 2	2442131	2352200	89931	3.68
	path 3	2442131	2352200	89931	3.68
fft	modref	1036666	1036398	268	0.03
	pointer	1016178	1003608	12570	1.24
	path 1	1016169	1003599	12570	1.24
	path 2	1016169	1003599	12570	1.24
	path 3	1016169	1003599	12570	1.24
	path 4	1016169	1003599	12570	1.24
clean	modref	83747	82442	1305	1.56
	pointer	83746	82402	1344	1.60
	path 1	87164	80189	6975	8.00
	path 2	87164	80189	6975	8.00
	path 3	87164	80189	6975	8.00
	path 4	87164	80189	6975	8.00
cachesim	modref	594387	594390	-3	-0.00
	pointer	594387	594390	-3	-0.00
	path 1	594387	594390	-3	-0.00
	path 2	594387	594390	-3	-0.00
	path 3	594387	594390	-3	-0.00
	path 4	594387	594390	-3	-0.00
dhrystone	modref	60010	60010	0	0.00
	pointer	56010	56010	0	0.00
	path 1	56010	56010	0	0.00
	path 2	56010	56010	0	0.00
	path 3	56010	56010	0	0.00
	path 4	56010	56010	0	0.00

Figure 6.7 Register promotion: store operations

Program	*analysis*	*without*	*with*	*difference*	*% removed*
water	modref	1035179	1035175	4	0.00
	pointer	1069711	1069645	66	0.01
	path 1	1069711	1069645	66	0.01
	path 2	1069711	1069645	66	0.01
	path 3	1069711	1069645	66	0.01
	path 4	1069711	1069645	66	0.01
indent	modref	71957	69140	2817	3.91
	pointer	72083	69141	2942	4.08
allroots	modref	11	11	0	0.00
	pointer	11	11	0	0.00
	path 1	11	11	0	0.00
	path 2	11	11	0	0.00
	path 3	11	11	0	0.00
	path 4	11	11	0	0.00
mybc	modref	243866	238488	5378	2.21
	pointer	243866	238488	5378	2.21
	path 1	243866	238488	5378	2.21
	path 2	245017	239639	5378	2.19
	path 3	245017	239639	5378	2.19
go	modref	20707586	20203423	504163	2.43
	pointer	20912109	20403336	508773	2.43
	path 1	20921613	20412840	508773	2.43
	path 2	20984093	20475320	508773	2.42
	path 3	20984093	20475320	508773	2.42
	path 4	20984093	20475320	508773	2.42
bison	modref	539941	539606	335	0.06
	pointer	539941	539606	335	0.06
	path 1	539941	539606	335	0.06
	path 2	539941	539606	335	0.06
	path 3	539941	539606	335	0.06
	path 4	539941	539606	335	0.06
jpeg	modref	2503016	2503022	-6	-0.00
	pointer	2503016	2503022	-6	-0.00

Figure 6.8 Register promotion: store operations (cont.)

Program	*analysis*	*without*	*with*	*difference*	*% removed*
gzip(enc)	modref	270393	212300	58093	21.48
	pointer	270392	198823	71569	26.47
	path 1	270392	198823	71569	26.47
	path 2	270392	198823	71569	26.47
	path 3	270392	198823	71569	26.47
	path 4	270392	198823	71569	26.47
gzip(dec)	modref	17565	17379	186	1.06
	pointer	17565	17233	332	1.89
	path 1	17565	17233	332	1.89
	path 2	17565	17233	332	1.89
	path 3	17565	17233	332	1.89
	path 4	17565	17233	332	1.89

Figure 6.9 Register promotion: store operations (cont.)

to a single store at the loop's exit, an effect that other optimization passes cannot achieve.

Program	*analysis*	*without*	*with*	*difference*	*% removed*
tsp	modref	113721	113721	0	0.00
	pointer	107937	107937	0	0.00
	path 1	107937	107937	0	0.00
	path 2	107937	107937	0	0.00
	path 3	107937	107937	0	0.00
	path 4	107937	107937	0	0.00
mlink	modref	27047909	27044130	3779	0.01
	pointer	25612630	25517809	94821	0.37
	path 1	25611769	25517130	94639	0.37
	path 2	25611658	25517019	94639	0.37
	path 3	25611658	25517019	94639	0.37
fft	modref	1253031	1252763	268	0.02
	pointer	1216662	1204103	12559	1.03
	path 1	1216662	1204103	12559	1.03
	path 2	1216662	1204103	12559	1.03
	path 3	1216662	1204103	12559	1.03
	path 4	1216662	1204103	12559	1.03
clean	modref	184158	183382	776	0.42
	pointer	181334	180429	905	0.50
	path 1	180023	179091	932	0.52
	path 2	180023	179091	932	0.52
	path 3	180023	179091	932	0.52
	path 4	180023	179091	932	0.52
cachesim	modref	1901534	1901536	-2	-0.00
	pointer	1900609	1900611	-2	-0.00
	path 1	1900609	1900611	-2	-0.00
	path 2	1900609	1900611	-2	-0.00
	path 3	1900609	1900611	-2	-0.00
	path 4	1900609	1900611	-2	-0.00
dhrystone	modref	62021	62021	0	0.00
	pointer	50021	50021	0	0.00
	path 1	46021	46021	0	0.00
	path 2	46021	46021	0	0.00
	path 3	46021	46021	0	0.00
	path 4	46021	46021	0	0.00

Figure 6.10 Register promotion: load operations

Program	*analysis*	*without*	*with*	*difference*	*% removed*
water	modref	2388037	2386374	1663	0.07
	pointer	1672710	1673879	-1169	-0.07
	path 1	1672710	1673879	-1169	-0.07
	path 2	1672710	1673879	-1169	-0.07
	path 3	1672710	1673879	-1169	-0.07
	path 4	1672710	1673879	-1169	-0.07
indent	modref	162389	147453	14936	9.20
	pointer	162742	147735	15007	9.22
allroots	modref	145	145	0	0.00
	pointer	141	141	0	0.00
	path 1	141	141	0	0.00
	path 2	141	141	0	0.00
	path 3	141	141	0	0.00
	path 4	141	141	0	0.00
mybc	modref	822343	817587	4756	0.58
	pointer	816780	812012	4768	0.58
	path 1	813374	808606	4768	0.59
	path 2	812682	807914	4768	0.59
	path 3	812682	807914	4768	0.59
go	modref	89793649	89224694	568955	0.63
	pointer	84594065	83945430	648635	0.77
	path 1	84500529	83851902	648627	0.77
	path 2	84514354	83865727	648627	0.77
	path 3	84514354	83865727	648627	0.77
	path 4	84514354	83865727	648627	0.77
bison	modref	553970	553731	239	0.04
	pointer	552828	552589	239	0.04
	path 1	551760	551519	241	0.04
	path 2	551760	551519	241	0.04
	path 3	551760	551519	241	0.04
	path 4	551760	551519	241	0.04
jpeg	modref	6596100	6596102	-2	-0.00
	pointer	6594785	6594787	-2	-0.00

Figure 6.11 Register promotion: load operations (cont.)

Program	*analysis*	*without*	*with*	*difference*	*% removed*
gzip(enc)	modref	906418	854592	51826	5.72
	pointer	897695	835906	61789	6.88
	path 1	897695	835906	61789	6.88
	path 2	897695	835906	61789	6.88
	path 3	897695	835906	61789	6.88
	path 4	897695	835906	61789	6.88
gzip(dec)	modref	136155	135968	187	0.14
	pointer	135823	135636	187	0.14
	path 1	135823	135636	187	0.14
	path 2	135823	135636	187	0.14
	path 3	135823	135636	187	0.14
	path 4	135823	135636	187	0.14

Figure 6.12 Register promotion: load operations (cont.)

Chapter 7

Conclusion

Our work has made contributions to the development of pointer analysis in two ways. First, we performed extensive experimental testing of pointer analysis. We provide performance numbers for both the analyzer and analyzed code. The size of our test suite is much larger than previous work. We experimentally justify the use of pointer analysis by showing that it can improve the performance of analyzed code. Performance improvement was especially dramatic in terms of the reduction in memory traffic. We also developed MOD/REF analysis as a benchmark against which to compare the results of pointer analysis. Second, we examined key questions and developed answers for them. In particular we explain why points-to analysis should be done instead of alias analysis. We also explain why explicit names should be used instead of representative names. Our experiments allowed us to quantitatively answer design questions concerning heap modeling and context information. Surprisingly, we found that using more accurate heap models did not significantly improve the performance of our benchmark code. We also show that context information can slightly improve the accuracy of pointer analysis. This is in contrast to the conclusion of Ruf's work. Furthermore, we show how context information can also slightly improve the performance of analyzed code. Unfortunately, we also show that context information is extremely expensive to compute in terms of time and space.

For anyone contemplating pointer analysis, our work shows that MOD/REF analysis is a good first step. It has many advantages. It is simpler. It is necessary for pointer analysis. It is safer. It brings much of the speedup that pointer analysis produces. It is also much less demanding in terms of time and space than pointer analysis. With the current state of computers, our pointer analyzer might be able to handle programs of about 100K lines of C code. This is good for research, but it will not be able to handle many industrial programs, which may have more than a million lines of code.

We also developed a new optimization pass, register promotion, that utilizes the information generated by pointer analysis or MOD/REF analysis. We show how it can reduce the memory traffic of programs.

Chapter 8

Future Work

Several areas were not explored or not fully explored by this thesis. These areas should be investigated more fully. Two of these areas, heap modeling and path information, were discussed in Section 2.

1. Heap modeling
2. Intraprocedural paths
3. Strong updates
4. Safety
5. Sources of approximation

8.1 Heap modeling

Three separate models for the heap were tested with our analyzer: splitting by `malloc` call site, using a single name, and splitting by user heap allocator call sites. Other models could be tried, for example: splitting along call paths and splitting by fields of heap allocated structures.

Generating heap regions is an expensive operation in the analyzer. Refinements to the heap model that increase the number of heap regions should be carefully considered. Heap regions are addressed regions, so they will appear on the region lists of all calls and pointer-based memory operations. This will expand the SSA name space and require more memory. Modeling the heap more accurately is not the ultimate goal. It may or may not produce more accurate pointer analysis. This in turn may or may not result in performance increases in the analyzed code. We must weigh all the potential improvements against the increased time and space required. This evaluation should be done by running various versions of the analyzer on the test suite.

8.2 Intraprocedural paths

No intraprocedural path information is used in the analyzer. Work should be done to see how it can be efficiently implemented, and what its impact on accuracy will be. One possibility is to record the basic block where an SSA name is defined. To test if two SSA names may interact at a memory operation, we need to find a path that passes through both definition points and reaches the basic block of the memory operation. This can be done by finding the strongly-connected components (SCC)s of the control-flow graph. Look for a path in the SCC graph that passes through the SCCs of the definition points and ends at the SCC of the use point. The existence of such a path must be determined quickly since it will be done repeatedly during propagation.

8.3 Strong updates

We have evidence from one experiment that strong updates do not improve the accuracy of our analyzer. However, perhaps the benefit of strong updates is being hidden by inaccuracies introduced by other sources of approximation. If we reduce the other sources of approximation, the benefit of strong updates may become evident.

8.4 Safety

The analyzer assumes that it can identify where addresses to new regions are generated. In the analyzer, the only place where these addresses should be generated are:

- load immediate of the address of a global
- add to a stack pointer
- add to a structure pointer
- heap-allocating call site

We are usually able to distinguish between an add operation that generates an address to a new region and an add to an array or heap pointer by the layout of memory regions in our analyzer's address space. The code in Figure 8.1 will illustrate how this works and a problem in this approach. The regions for this code can be seen at the bottom of the figure (note that `p` does not have a region since it is allocated to a register). Regions are shown with their address above their top left corner. The code in this first section would produce the `ILOC` in Figure 8.2. Register `r0` will be initialized with address 0, since this is the base of the stack frame. Thus, we recognize that the `iADDI` generates an address to a new region because the resulting address, 4, is the start of region `@_c.b`. On the other hand, in the `iADD` operation, `r4` will contain the address 4, and `r5` will contain 4. When these are added together, the result, 8, is not the base of any region, so we do not add this address to `r6`. Instead, we assume that 4 is the address of an array region, so when we add to it we generate the same address, 4. Thus, 4 will be inserted in the list of addresses in r6.

Problems with this process can occur when we use type casts. For the code in the second section of Figure 8.1, the analyzer will incorrectly conclude that the first store through `p` stores to `@_c.b`, and the second store through `p` stores to `@_c.a`. This occurs because the analyzer does not understand how to deal with type casts. All stores through `p` should be to the same region. Unfortunately, the underlying structure that was inherited through the type cast puts a region, `@_c.b`, in the middle of the array pointed to by `p`. The first store happens to hit the base of this region, so we conclude that it is a store to `@_c.b`. When `p` is incremented, this lands in the middle of region `@_c.b`. Since this is not the base of a region, the analyzer assumes this is an array increment and sets `p` to its original address. Thus, `p` points to region `@_c.a` after the increment. Further work needs to be done to ensure that this case is handled correctly.

8.5 Sources of approximation

It would be very helpful for further work in pointer analysis to understand the frequency of each source of approximation. Unfortunately, this information may be very difficult to generate (see Section 4.5).

```
struct foo {
  int a,b[2];
};

void f() {
  int        *p;
  struct foo c;

/* good addressing */

  p=c.b;
  p[1]=3;

/* bad addressing */

  p=(int *) &c;
  p[1]=3;          /* analyzer thinks this stores to c.b */
  p+=2;
  p[0]=3;          /* analyzer thinks this stores to c.a */
}
```

f's function regions

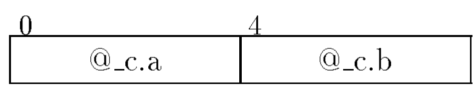


Figure 8.1 Good and bad address creation

```
        9       iADDI   4 r0 => r3              # get address of f_c
        9       i2i     r3 => r2
        10      i2i     r2 => r4
        10      iLDI    4 => r5
        10      iADD    r4 r5 => r6
        10      iLDI    3 => r7
        10      iSTor   ! 4 0 r6 r7
```

Figure 8.2 ILOC code with no address creation problems

Appendix A

ILOC

In this section we give a quick description of our intermediate language, `ILOC`, which should be sufficient for understanding this thesis. A more complete description is given in our documentation for the MSCP[3].

In Figure A.1, a short piece of C code is given with its corresponding `ILOC` code. `ILOC` is an assembly-like language with an unlimited number of symbolic registers. Registers always start with the letter "r". Functions in `ILOC` always start with a `FRAME` declaration. In our example, the label for the function being declared is: `_swap`. Labels always start with an "_" or an "L". The `_swap` label is followed by the number of the source line in C code that created this operation. Since a `FRAME` operation is not executable code, it has no associated source line, so this number is `0`. Following the `FRAME` opcode comes the size of the stack frame for this function and the registers for the parameters passed into the function. The first two parameters are non-user parameters. The first parameter is the stack pointer. The second is necessary for the implementation of `ILOC`. The types of the parameters follow the registers. The types for `ILOC` are shown in Figure A.2. In our example, the function has four parameters, two of which are user parameters. All of the parameters are of integer type. Functions in `ILOC` are terminated with a `RTN`, return, or `HALT` operation.

Functions and basic blocks are always given at least one label. There are five labels in this example: `_swap`, `L3_swap`, `L1_swap`, `L2_swap`, and `_g`. The label `_swap` is the label for the function and the entry block to the function. The next three labels are also for basic blocks of the function. The last label `_g` is for a data segment. Data segments are made up of `DATA` statements. `DATA` statements allocate global memory. The one letter lower-case prefix tells the type of the memory being allocated. The syntax of a `DATA` statement is:

`DATA` *initial_value repetition_factor*

The *repetition_factor* tells the number of copies of the specified data type for which to allocate memory.

Calls, loads, and stores can be "tagged" with the memory locations they may modify or reference. Tags occur in two types, base and group. Base tags represent a

C source code

```
int g;

void swap(int *a, int *b) {
  int temp;

  if (*a) {
    temp=*a;
    *a=*b;
    *b=temp;
  } else
    g=3;
}
```

ILOC code

```
_swap:          0       FRAME   0 => r0 r1 r2 r3 [ i i i i ]
                6       i2i     r2 => r5
                6       iPLDor  @! 4 0 r5 => r6
                6       iLDI    0 => r7
                6       iCMPeq  r6 r7 => r25
                6       BR      L1_swap L3_swap r25
L3_swap:        7       i2i     r2 => r9
                7       iPLDor  @! 4 0 r9 => r10
                7       i2i     r10 => r4
                8       i2i     r3 => r11
                8       iPLDor  @! 4 0 r11 => r12
                8       iPSTor  @! 4 0 r9 r12
                9       i2i     r3 => r14
                9       i2i     r4 => r15
                9       iPSTor  @! 4 0 r14 r15
                10      JMPl    L2_swap
L1_swap:        11      iLDI    _g => r20
                11      iLDI    3 => r21
                11      iSSTor  @_g_0 4 0 r20 r21
L2_swap:        12      RTN     r0
_g:     bDATA 0 4
        ALIAS @_writable_globals [ @_g_0 ]
        iSGLOBAL        0       @_g_0   _g
```

Figure A.1 Sample ILOC function

prefix	**description**	**size (bytes)**
`b`	byte or character	1
`w`	word	2
`i`	integer	4
`f`	single-precision floating-point	4
`d`	double-precision floating-point	8
`c`	single-precision complex	8
`q`	double-precision complex	16

Figure A.2 `ILOC` types

region of memory (*e.g.*, an array, a local variable, a string, *etc.*). Group tags represent a collection of base tags. Tags always start with an "@". The group tag "@!" is used to represent the collection of all base tags. For operations that have a type, the type is specified by a prefix attached to the opcode. For example, `i2i` is an integer copy (read: integer to integer). Memory operations come in two varieties, scalar and pointer. Scalar memory operations have a "S" following their type prefix, while pointer-based memory operations have a "P". There is no distinction between the actions of a scalar and a pointer-based memory operation. Their purpose is to aid the accuracy of the analysis of our optimization passes. Scalar memory operations denote operations on a known non-aggregate (*i.e.*, non-heap and non-array) memory location. The memory location being acted on is specified by the base tag in the operation. Pointer memory operations are used for all other memory operations. The tags used for these operations may be group tags or base tags for aggregate memory locations. Using our nomenclature, the opcode `iPSTor`, stands for an integer pointer store. The opcode `dSLDor` stands for a double-precision floating point scalar load. In our example, the line:

```
11      iSSTor  @_g_0 4 0 r20 r21
```

is an integer scalar store to base tag `@_g_0`. The base register is `r20` and the value being stored is in `r21`. The `ALIAS` statement is used to add base tags to a group tag. The syntax is:

```
ALIAS @_group_tag [@base_tag1 @base_tag2 ... ]
```

The GLOBAL statement is used to declare information about a tag for global memory. The one letter lower-case prefix tells the type of the memory. If this is followed by an "S", then the memory is scalar. Otherwise, it is non-scalar (*i.e.*, it is an array or heap). This is followed by the offset, the tag itself, and a global label. The global label plus the offset is the address of the base of the memory region. The STACK statement is the analog of the GLOBAL statement for stack memory. The label in a STACK statement is the label of the function that creates the stack memory.

Figure A.3 gives a chart of of ILOC opcodes roots and a description of their function.

Opcode	Description
LDI	load immediate
LDor	load
STor	store
ADD	add
ADDI	add immediate
SUB	subtact
MUL	multiply
DIV	divide
CMP	compare
2	move
RTN	return
JSRl	explicit call
JSRr	indirect call
BR	branch
JMPl	jump to a label
JMPr	computed jump
FRAME	declaration for a function
GLOBAL	declaration for a global tag
STACK	declaration for a local tag
DATA	declaration for global memory
ALIAS	declaration for a group tag

Figure A.3 `ILOC` opcode roots

Appendix B

Supplementary Results

Our compiler environment is constantly under development. Many improvements, including a better register allocator, were installed while this thesis was being written. Some of the experiments testing the performance of analyzed code were re-run with these improvements. The results of these experiments are shown in Figures B.1, B.2, B.3, B.4, B.5, and B.6. These results were produced from code analyzed with a MEDIUM heap model and are directly comparable to the results in Section 5.2.2.

B.0.1 Cache Results

We also ran our analyzed programs with two models that contained a cache simulator. In the first model we simulated a single-level 2KB, 2-way set-associative write-back cache with a line size of 32 bytes. Figure B.7 shows the results for the single-level cache model. The "% of total" line shows the percentage of operations in the program that access the cache. The "miss %" line shows the percentages of accesses that are not serviced by the cache. Increasing analysis usually decreased the percentage of cache accesses. This is to be expected since our analysis frequently allows memory operations to be removed. However, this also reduced the cache locality and thus increased the miss percentage. In the second model we simulated a two-level cache with a 1KB direct-mapped, write-through L1 cache, and a 64KB 4-way set-associative, write-back L2 cache, each with a line size of 16 bytes. Figures B.8 and B.9 show the results for the two-level cache model. The results for this model follow a format that is similar to the results for the single-level model, however, results for each level of the cache are presented. The "% of total" line for the L2 cache is the percentage of operations that access the L2 cache. This includes L1 cache misses and L1 cache write-throughs. As in the single-level cache model, increasing analysis usually decreased the percentage of L1 cache accesses. However, unlike the single-level model this had a varied effect on the L1 cache misses. Both increases and decreases in miss percentages were seen. L2 cache results were unpredicatable. Increases and decreases in both the percentage of cache accesses and percentage of cache misses were seen when the level of analysis was increased.

Program	Original	Modref	Path 0	Path 1	Path 2
tsp	729899	657238	651454	651454	651454
mlink	131733946	123952927	121309799	121308656	121308224
fft	13647090	12642655	12557372	12557363	12557363
clean	1080339	1021799	1015224	1017177	1017177
cachesim	11960929	11589353	11587392	11587392	11587392
dhrystone	602196	588171	556171	552171	552171
water	14584452	13379663	12603865	12603865	12603865
indent	906980	828188	828252		
allroots	1039	1021	1011	1011	1011
bc	5464782	5430853	5419956	5415399	5410282
go	495428725	489854354	478821979	478590106	478565722
bison	3848257	3312990	3310092	3306882	3306882
jpeg	36876573	36935016	36933699		
gzip(enc)	5725701	5680328	5648030	5648030	5648030
gzip(dec)	990565	987510	986828	986828	986828

Figure B.1 Supplementary results: total operations

Program	% eliminated			
	Original	Modref	Path 1	Path 2
tsp	10.7	0.9	0.0	0.0
mlink	7.9	2.1	-0.0	-0.0
fft	8.0	0.7	-0.0	-0.0
clean	6.0	0.6	0.2	0.2
cachesim	3.1	0.0	0.0	0.0
dhrystone	7.6	5.4	-0.7	-0.7
water	13.6	5.8	0.0	0.0
indent	8.7	-0.0		
allroots	2.7	1.0	0.0	0.0
bc	0.8	0.2	-0.1	-0.2
go	3.4	2.3	-0.0	-0.1
bison	14.0	0.1	-0.1	-0.1
jpeg	-0.2	0.0		
gzip(enc)	1.4	0.6	0.0	0.0
gzip(dec)	0.4	0.1	0.0	0.0

Figure B.2 Supplementary results: total operation removal percentages

Program	Original	Modref	Path 0	Path 1	Path 2
tsp	51047	51046	51046	51046	51046
mlink	2114631	2111388	2025124	2025120	2025120
fft	1036396	1036396	1003607	1003598	1003598
clean	76297	56010	55970	55970	55970
cachesim	685931	594390	594390	594390	594390
dhrystone	60007	60007	56007	56007	56007
water	1034246	1034241	1034176	1034176	1034176
indent	68599	64421	64421		
allroots	11	11	11	11	11
bc	238850	232440	232440	232440	232440
go	18586094	18363963	18373173	18373173	18373173
bison	539964	539494	539494	539494	539494
jpeg	2499954	2499960	2499960		
gzip(enc)	216990	212300	198823	198823	198823
gzip(dec)	17765	17376	17230	17230	17230

Figure B.3 Supplementary results: stores

Program	% eliminated			
	Original	Modref	Path 1	Path 2
tsp	0.0	0.0	0.0	0.0
mlink	4.2	4.1	-0.0	-0.0
fft	3.2	3.2	-0.0	-0.0
clean	26.6	0.1	0.0	0.0
cachesim	13.3	0.0	0.0	0.0
dhrystone	6.7	6.7	0.0	0.0
water	0.0	0.0	0.0	0.0
indent	6.1	0.0		
allroots	0.0	0.0	0.0	0.0
bc	2.7	0.0	0.0	0.0
go	1.1	-0.1	0.0	0.0
bison	0.1	0.0	0.0	0.0
jpeg	-0.0	0.0		
gzip(enc)	8.4	6.3	0.0	0.0
gzip(dec)	3.0	0.8	0.0	0.0

Figure B.4 Supplementary results: store removal percentages

Program	Original	Modref	Path 0	Path 1	Path 2
tsp	166549	113720	107936	107936	107936
mlink	29835970	24336975	22788457	22787292	22787181
fft	2012054	1252753	1204097	1204097	1204097
clean	196710	151540	148784	147473	147473
cachesim	2102248	1901536	1900611	1900611	1900611
dhrystone	68033	62018	50018	46018	46018
water	3034509	2369636	1610997	1610997	1610997
indent	201035	137955	138188		
allroots	163	145	141	141	141
bc	827239	799295	793562	790156	785039
go	90119721	85852361	80550739	80421579	80341350
bison	942318	553621	552479	551409	551409
jpeg	6526461	6586920	6585605		
gzip(enc)	901537	844628	825942	825942	825942
gzip(dec)	138317	135962	135629	135629	135629

Figure B.5 Supplementary results: loads

Program	% eliminated			
	Original	Modref	Path 1	Path 2
tsp	35.2	5.1	0.0	0.0
mlink	23.6	6.4	-0.0	-0.0
fft	40.2	3.9	0.0	0.0
clean	24.4	1.8	-0.9	-0.9
cachesim	9.6	0.0	0.0	0.0
dhrystone	26.5	19.3	-8.7	-8.7
water	46.9	32.0	0.0	0.0
indent	31.3	-0.2		
allroots	13.5	2.8	0.0	0.0
bc	4.1	0.7	-0.4	-1.1
go	10.6	6.2	-0.2	-0.3
bison	41.4	0.2	-0.2	-0.2
jpeg	-0.9	0.0		
gzip(enc)	8.4	2.2	0.0	0.0
gzip(dec)	1.9	0.2	0.0	0.0

Figure B.6 Supplementary results: load removal percentages

Program		Original	MOD/REF	Path 0
tsp	% of total	29.81	25.07	24.40
	miss %	0.32	1.98	1.39
mlink	% of total	24.25	21.34	20.45
	miss %	17.15	19.19	21.05
fft	% of total	22.34	18.11	17.58
	miss %	9.63	12.25	12.24
clean	% of total	25.27	20.31	20.17
	miss %	18.45	22.62	22.00
dhrystone	% of total	21.26	20.75	19.06
	miss %	0.01	0.01	0.02
water	% of total	27.90	25.44	20.99
	miss %	3.50	3.79	5.28
indent	% of total	29.70	24.41	24.44
	miss %	5.58	5.05	9.21
allroots	% of total	16.75	15.28	15.03
	miss %	6.90	7.69	7.24
bc	% of total	19.51	19.00	18.93
	miss %	4.03	5.73	5.78
go	% of total	21.94	21.27	20.66
	miss %	32.47	39.15	39.09
bison	% of total	38.52	32.99	32.99
	miss %	2.63	1.65	1.69
jpeg	% of total	24.48	24.60	24.60
	miss %	16.60	16.43	16.38
gzip (enc)	% of total	19.54	18.61	18.14
	miss %	30.02	32.54	32.52
gzip (dec)	% of total	15.76	15.53	15.50
	miss %	5.52	5.69	5.85

Figure B.7 Single-level cache statistics

Program	cache level		Original	MOD/REF	Path 0
tsp	1	% of total	29.81	25.07	24.40
		miss %	18.35	7.38	7.67
	2	% of total	6.91	2.65	2.67
		miss %	2.61	7.49	7.50
mlink	1	% of total	24.25	21.34	20.45
		miss %	29.76	31.72	35.80
	2	% of total	8.39	7.94	8.61
		miss %	2.00	2.00	2.09
fft	1	% of total	22.34	18.11	17.58
		miss %	11.36	13.51	12.15
	2	% of total	4.44	4.46	3.94
		miss %	18.91	20.26	23.07
clean	1	% of total	25.27	20.31	20.17
		miss %	23.65	28.29	27.99
	2	% of total	9.04	7.99	7.79
		miss %	2.87	3.43	3.57
dhrystone	1	% of total	21.26	20.75	19.06
		miss %	3.14	6.57	7.56
	2	% of total	2.00	4.09	4.68
		miss %	0.17	0.08	0.07
water	1	% of total	27.90	25.44	20.99
		miss %	12.13	12.92	16.24
	2	% of total	5.82	6.96	7.21
		miss %	0.33	0.29	0.30
indent	1	% of total	29.70	24.41	24.44
		miss %	11.15	11.56	12.35
	2	% of total	6.06	5.22	6.01
		miss %	2.08	2.60	2.46
allroots	1	% of total	16.75	15.28	15.03
		miss %	13.22	10.26	9.87
	2	% of total	2.41	1.76	1.68
		miss %	60.00	88.89	88.24
bc	1	% of total	19.51	19.00	18.93
		miss %	10.96	10.34	11.42
	2	% of total	2.76	3.04	3.29
		miss %	0.98	0.93	0.87

Figure B.8 Two-level cache statistics

Program	cache level		Original	MOD/REF	Path 0
go	1	% of total	21.94	21.27	20.66
		miss %	39.21	43.93	44.46
	2	% of total	9.64	10.53	10.39
		miss %	2.17	3.79	2.27
bison	1	% of total	38.52	32.99	32.99
		miss %	21.95	2.57	2.38
	2	% of total	12.17	5.21	5.19
		miss %	2.03	5.52	5.56
jpeg	1	% of total	24.48	24.60	24.60
		miss %	19.33	19.26	18.92
	2	% of total	8.11	8.05	8.09
		miss %	0.11	0.11	0.12
gzip (enc)	1	% of total	19.54	18.61	18.14
		miss %	33.56	36.91	36.82
	2	% of total	7.55	8.06	7.81
		miss %	6.42	5.99	6.21
gzip (dec)	1	% of total	15.76	15.53	15.50
		miss %	9.57	9.79	10.27
	2	% of total	2.11	2.13	2.19
		miss %	15.71	15.73	15.27

Figure B.9 Two-level cache statistics (cont.)

Bibliography

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

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

[3] Preston Briggs. The massively scalar compiler project. Technical report, Rice University, 1994.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[5] Preston Briggs, Rob Shillner, and Taylor Simpson. Dead code elimination. Technical report, Rice University, June 1993. Available via anonymous ftp.

[6] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, March–December 1993.

[7] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[8] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[9] Steve Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, Department of Computer Science, 1992.

[10] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[11] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.

[12] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[13] 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 1985.

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

[15] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[16] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994.

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

[18] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. *SIGPLAN Notices*, 28(6):90–99, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[19] J. T. Hecht, Y. Wang, B. Connor, S. H. Blanton, and S. P. Daiger. Nonsyndromic cleft lip and palate: No evidence of linkage to hla of factor 13a. *American Journal of Human Genetics*, 52:1230–1233, 1993.

[20] Joseph Hummel, Laurie J. Hendren, and Alexander Nicolau. A general data dependence test for dynamic, pointer-based data structures. *SIGPLAN Notices*, 29(6):218–229, June 1994.

[21] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 8(6):66–74, January 1982.

[22] Robert W. Cottingham Jr., Ramana M. Idury, and Alejandro A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[23] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, Florida, January 1991.

[24] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, June 1992.

[25] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *SIGPLAN Notices*, 23(7):21–34, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[26] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus analysis in humans. *Proceedings of the National Academy of Sciences of the U.S.A.*, 81:3443–3446, 1984.

[27] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

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

[29] E. Morel and C. Renvoise. Global optimization by supression of partial redundancies. *Communications of the ACM*, 22(1):96–103, January 1979.

[30] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[31] Eugene W. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, Williamsburg, Virginia, January 1981.

[32] Erik Ruf. Context-insensitive alias analysis reconsidered. *SIGPLAN Notices*, 30(6):13–22, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[33] Alejandro A. Schäffer, Sandeep K. Gupta, K. Shriram, and Robert W. Cottingham Jr. Avoiding recomputation in linkage analysis. *Human Heredity*, 44:225–237, 1994.

[34] Taylor Simpson. Global value numbering. Technical report, Rice University, 1994. Available via anonymous ftp.

[35] M.N. Wegman and F.K. Zadeck. Constant propogation with conditional branches. Technical Report CS-89-36, Brown University, 1989.

[36] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. *SIGPLAN Notices*, 30(6):1–12, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*