FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

INTER-PROCEDURAL GLOBAL VARIABLE DATA DEPENDENCE DETECTION

By

PHALGUNA RUPANAGUDI

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the requirements for graduation with
Honors in the Major

Degree Awarded:
Spring,2018

The members of the Defense Committee approve the thesis of Phalguna Rupanagudi defended on April 24, 2018 (Signatures on file with Florida State University Honors Program).

David Whalley
Thesis Director

Michael Mascagni
Committee Member

Timothy Stover
Outside Committee Member

ACKNOWLEDGMENTS

My sincerest thank you goes out to Dr. Mascagni, and Dr. Stover for their guidance and eagerness to help over the past few semesters. Many thanks go out to Dr. Whalley for taking me into his compilers group and helping me mature as a researcher and a student. Finally, thanks to everyone in the compilers group, especially Ryan Baird, for helping me in every step of the process, even with helping configure my vimrc file.

TABLE OF CONTENTS

ABSTRACT

Compiler optimization is an area of research in computer science dealing with improving performance of an executable, given the source code. Performance improvement can be measured by faster execution time, or less memory usage. When implementing compiler optimizations, it is important for the compiler to ensure that the functionality of the program does not change. The purpose of this project is to perform analysis to determine whether certain optimizations can be performed.

This project deals with the inter-procedural detection of data dependencies of global variables. If we can determine that for a given function which global variables can be accessed, then it may be possible to overlap the execution of the function with other functions that invoke it. they do not have overlapping dependencies, they may possibly be scheduled by the compiler in the optimal order.

The program will conservatively determine points-to sets; a set of global variables which may be referenced, for every function in the program. The two stage process includes the initial intra-procedural computation of points-to sets. The subsequent stage, performed outside of the compiler, will traverse the function call-graph for the program being analyzed and propagate the pointer information up the graph.

CHAPTER 1

INTRODUCTION

1.1 Background

Compiler optimizations, carried out by statically by an optimizing compiler, are intended to improve performance of programs. Performance can be measured by run time, memory use, or power consumption. The most popular optimizations involve decreasing execution time of programs and such optimization usually involve manipulating the original code for better performance. However, not all optimizations can be applied on each program, so the compiler must initially detect which techniques are allowed to be performed before applying the optimizations. This project will explore an initial detection technique for memory dependencies, which can be used to determine whether certain compiler optimizations, such as parallelization, are implementable.

1.2 Motivation

Multiprocessing, the method of using multiple processors or machines to carry out a single task, has significantly increased computing power through parallelization. However, since most programs are sequential in nature, we must ensure that parallelization doesn't change the semantics of the program. Much research has been done on instruction level parallelism, but DDD(Data Dependence Detection) explores possible exploitation of function level parallelism. Compared to sequential execution of programs, function level parallelization can potentially achieve a speed up of up to the number of processors in the system [4].

1.3 Purpose

The purpose of DDD is to conservatively determine memory reads and memory writes at the function level. Detecting potential data hazards is a necessary step in determining whether two functions can be parallelized. If any given two functions have overlapping memory dependencies then we can conclude that they cannot be parallelized, but if they don't have any overlapping dependencies, they are possible candidates for parallelization.

1.4 Types of Data Dependencies Detected

DDD will assist in determining the read and write dependencies in memory for each function in the program. The compile time analysis will determine all of the possible memory locations that can be written to or read from by a single function and call graph analysis will propagate all read and write information through functions calls. By detecting memory dependencies before performing optimization, the compiler can detect potential data hazards which could cause an optimization to generate invalid code.

Listing 1.1: Read After Write

```
int a, b, c;

int main(){
  foo(); ←
  bar(); ←
}

void foo() {
  a = 10;
  foo2();
}

void bar() {
  if(a == 10)
    b=101;

}
void foo2() {
  if(b)
    ...
}
```

This is an example of a Read after Write dependency. Global variable *a* is being written in function *foo* and being read from in function *bar*.

Listing 1.2: Write after Read

```
int a, b, c;

int main(){
  bar();←
  foo();←
}

void foo() {
  a = 10;
  foo2();
}

void bar() {
  if(a == 10)
    ...

}
void foo2() {
  if(b)
    ...
}
```

This code snippet is an example of a write after read dependency, when *bar* is called before *foo*.

This example also illustrates the need for call graph propagation of memory accesses. Since *foo2* reads from *b* and *foo2* is called from *foo*, we must declare that *foo* writes to *a* and reads from *b*.

CHAPTER 2

BACKGROUND

2.1 Sensitivities

Context Free Sensitivity. An intra-procedural analysis for each function is only executed once, so calling context will not be considered. More specifically, the compiler side analysis doesn't analyze a function each time it is called. However, for each function call the points-to set information for each parameter is stored and used in the call graph analysis. The consequence of this method is a more conservative list of memory accesses in a program. Consider this example:

Listing 2.1: Context Free Example

```
int a,b;

int *x;

void foo(){
  *x = 1;
}

int main(){
  x = &a;
  foo();

  x = &b;
  foo();
}
```

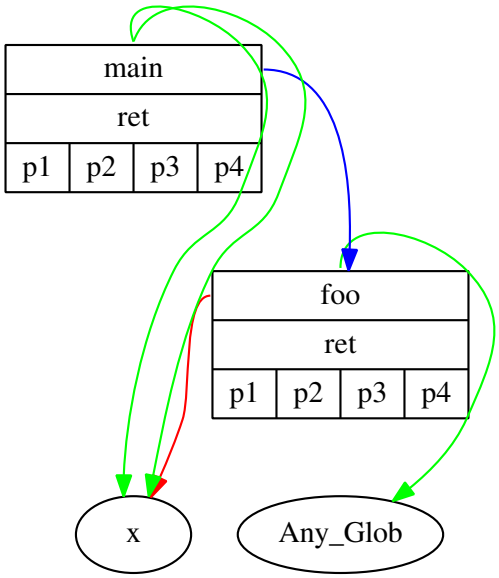

Figure 2.1: Intra-procedural analysis of Listing 2.1

Since the analysis of function *foo* is done only once, the analysis of functions shows that it will write to(designated by the green arrow) an unspecified global, which provides a more conservative analysis.

2.2 Path Sensitivity

In the compile time analysis, we employ path sensitivity, the principle of taking into account the order of statements in the program. This is made possible by the traversal of basic blocks in the intra-procedural analysis. By traversing all basic blocks, we ensure that we take every possible path through the function.

2.3 Popular Techniques Referenced

2.3.1 Andersen Style Pointer Analysis (ASPA)

Andersen-Style Pointer Analysis [2] is among the most popular pointer analysis techniques used in the industry. It is flow insensitive; all instructions are considered regardless of control flow, and context insensitive; assumes one behavior at all call sites.

Table 2.1: Andersen Style Pointer Analysis

PTS(A) = PointsToSet(A)

Instruction	Constraint	Meaning of Constraint
A = &B	A $\supseteq$ {B}	B $\in$ PTS(A)
A = B	A $\supseteq$ B	PTS(A) $\supseteq$ PTS(B)
A = *B	A $\supseteq$ *B	PTS(A) $\supseteq$ PTS(*B)
$*$A = B	*A $\supseteq$ B	PTS(*A) $\supseteq$ PTS(B)

1. Iterate through the input program and collect constraints as referenced in the table
2. Build a Directed Graph G = <V,E> where V is the set of pointers and E is the set of edges between pointers
3. Edge A $\rightarrow$ B iff one of the following is true:
   - PTS(A) $\subseteq$ PTS(B)
   - A $\in$ PTS(v) and PTS(*v) $\subseteq$ PTS(B)
   - A $\subseteq$ PTS(*v) and B $\in$ PTS(v)

Using the first two basic constraints from Andersen Style Pointer Analysis, we build our modified version of pointer analysis. The motivation to build points to set information for each register comes from the the analysis.

2.3.2 Iterative Data Flow Analysis (IDFA)

For both the compile time analysis, and call graph analysis we use the popular iterative data flow analysis technique [1]. IDFA involves three main basic principles:

1. Assign an *inState* and an *outState* for each designated *block*. In the compile analysis a block is defined to be a basic block, and in the call graph analysis a block is defined to be a function.
2. Apply the transfer function on each block and collect the *outState*, which is a function of the *inState*.
3. Propagate the *inState* and *outState* along control flow edges until all *inState* and *outState* converge.

CHAPTER 3

IMPLEMENTATION

3.1 Technologies

3.1.1 VPO

The Very Portable Optimizer (VPO), a C language compiler system, will be used for the compiler side analysis of this project. VPO uses nontraditional code generation approaches which prove to be advantageous for development and portability purposes. As opposed to performing optimizations on multiple code representations,VPO uses RTL's (Register Transfer Lists) in all phases of optimization. This is beneficial in reordering of optimizations. RTL's represent legal machine language instructions, which are easy to understand and manipulate [3]. For example,

$$RTL:\ r[1] = r[2] + r[3]$$

$$MIPS:\ add\ \$t1, \$t2, \$t3$$

these two instructions represent the same machine language instruction.

VPO stores all RTL's in a data structure that groups them by basic blocks. Basic blocks are sets of instructions such that the only entrance into the block is the first instruction, and the only exit from the block is the last instruction. By grouping the RTL's by basic blocks, VPO provides a mechanism for traversing functions through their natural control flow. Furthermore, in the data structure, pointers to all predecessors of each basic block are stored. This data structure is extremely important for DDD as traversing the control graph is necessary for ensuring flow sensitivity [5].

3.2 Compile Time Analysis

3.2.1 Transfer Function

The transfer function for the compile time analysis is a modified version of the Andersen's constraint collection method.

```
//r[a],r[b] are registers
//M[a] is value at  memory address a

def CollectConstraints:
   if( r[a] = r[b] ):
      PTS(a) = PTS(b)
   else if( r[a] = M[g] ): // a load instruction
      PTS(a) = SetOfAllGlobals
   else if( r[a] = g ):
      PTS(a) = {g}
```

Figure 3.1: Compile Time Transfer Function

Table 3.1: DDD vs ASPA

Type	DDD	ASPA
1	$r[a] = g$	$A = \&B$
2	$r[a] = r[b]$	$A = B$
3	$r[a] = M[g]$	$A = *B$

The instructions $r[a] = g$ and $A = \&B$ are logically equivalent. Within the back end of the compiler, the variable g is a label which represents memory that was allocated. The label g can be viewed as a constant pointer.

The type 2 constraints represent direct assignments of variables. In this case the points to set of the right hand side is directly assigned to the point to set of the left hand side. For the cases where there are two registers on the left hand side such as $r[1] = r[2] + r[3]$, the union of the points to sets is assigned to the left hand side.

The modification of the ASPA comes with the two complex constraints that he proposes. In our case, we do not follow pointer information through memory, so we conservatively declare that the points-to set information resulting from a load is the set of all globals.

3.2.2 Compile Time Psuedocode

The following code illustrates the compile time analysis algorithm used in DDD. The initial *do-while* loop illustrates the iterative data flow analysis technique implemented. The subsequent pass through the RTL's creates the call graph with the possible stores and loads of a function.

```
do{
   CHANGE = False
   for each Basic Block B in function F:
   {
      for each register r:
         for each predecessor P of B:
            Inset[B][r] = Union(Inset[B][r], Outset[P][r])

      COPY = Inset
      for each RTL R in B:
         COPY = CollectConstraints(R)

      for each register r:
         if COPY[B][r] != Outset[B][r]:
            CHANGE = True
            Outset[B][r] = Copy[B][r]
   }
}while(CHANGE)

for each Basic Block B in function F:
{
   for each RTL R in Basic Block B:
   {
      if is_a_store(R): //R[r[a]] = r[b]
         print F + "stores to" + PTS(a)

      if is_a_load(R): //r[a] = R[r[b]]
         print F + "loads from" + PTS(b)

   }

}
```

Figure 3.2: Compile Time Pseudocode

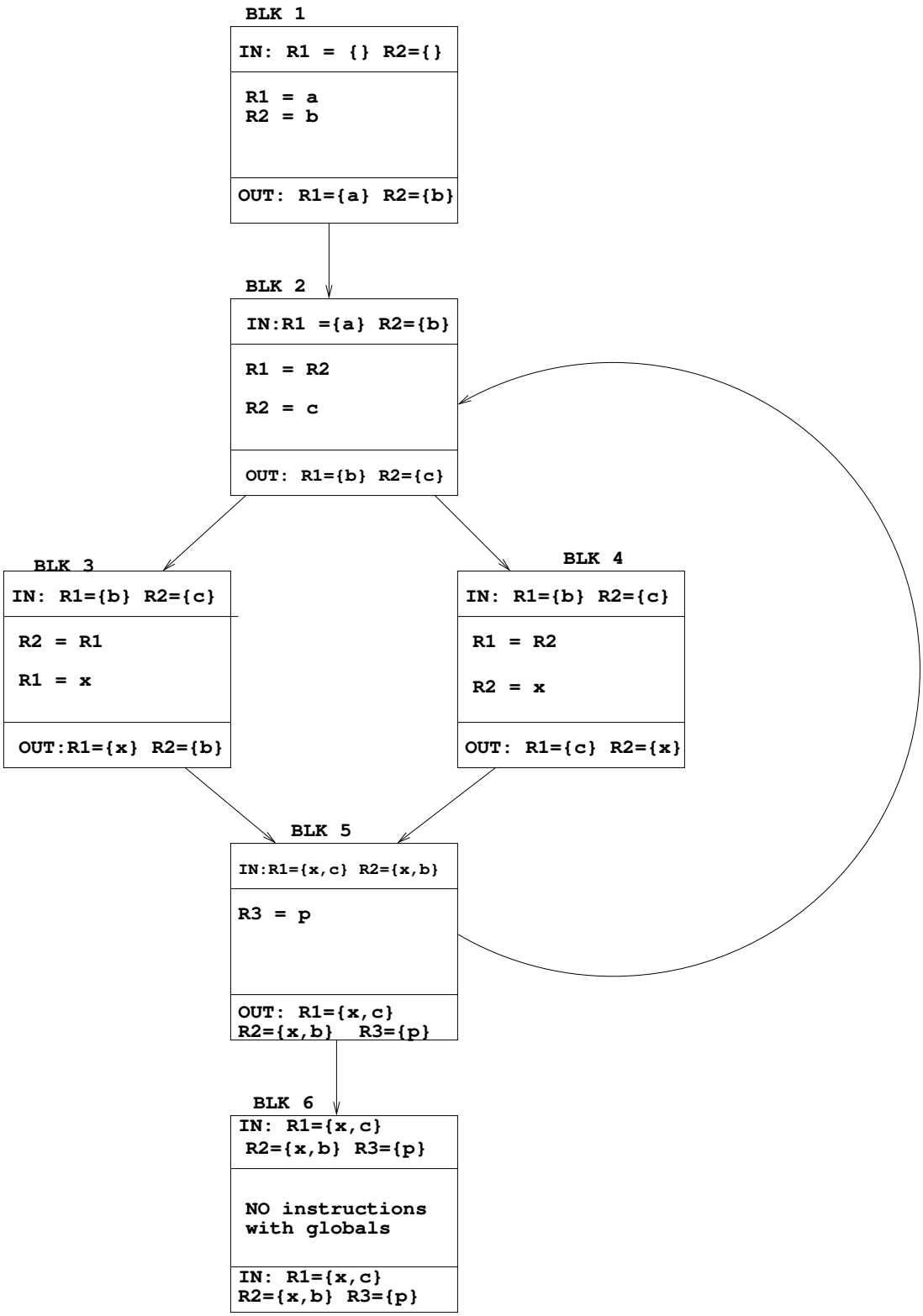

Figure 3.3: First iteration of loop

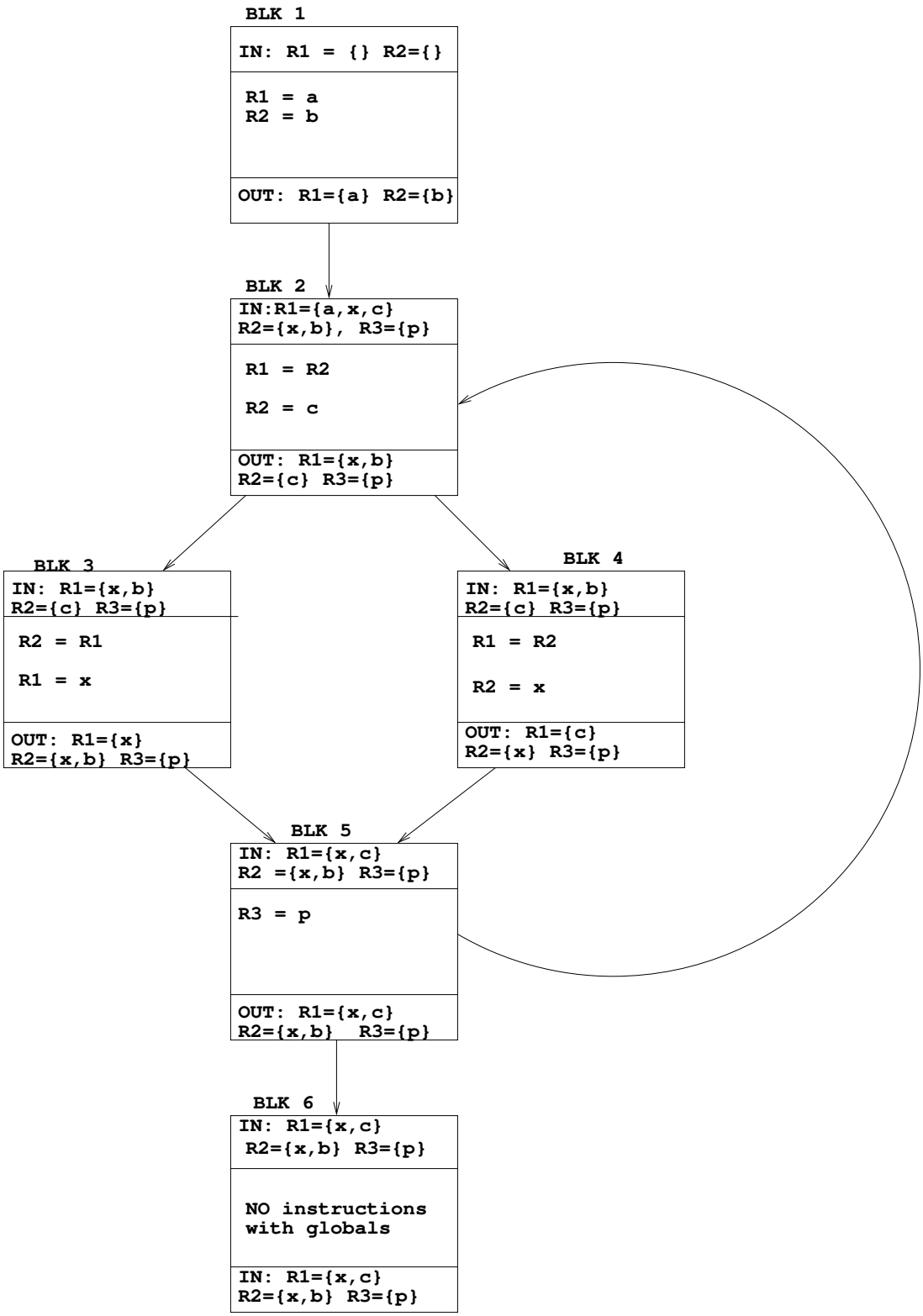

Figure 3.4: Second iteration of loop

Figures 3.2 and 3.3 are an illustration of the propagation of the points to sets through the basic blocks of a function. This specific part of the function contains a loop and six basic blocks. Only instructions that affect points to set data are shown for each block. Notice the change in the points to sets for block 2 between the two iterations. Blocks 1 and 5 are both predecessors of block 2, and when the in sets of block 2 were calculated in the first iteration, the out sets of block 5 were empty. In the second iteration, block 2 reflects the changes made in the loop. The third iteration of the algorithm will show no changes, meaning all propagation was completed in the second iteration.

3.3 Call Graph Analysis

Call graph analysis takes the output of the compile time analysis; the call graph, and propagates the points-to set information through function calls. This step is independent of the compile time analysis since we chose to use context free pointer analysis. Since each function is only analyzed once in VPO, we may not have information about the memory dependencies about the callee functions at call time.

In order to conservatively determine what memory dependencies a function has, we must also consider the memory dependencies of all of the callee functions. Consider the call graph from listing 2.1:

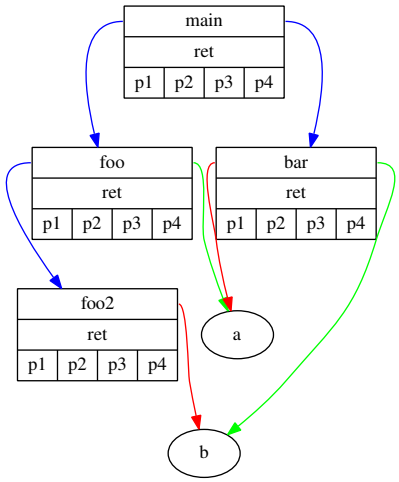

Figure 3.5: Pre-Call Graph Analysis

Function *foo* does not directly read from global b, but since *foo2* reads from b, we must declare that *foo* and *bar* have overlapping memory dependencies. In order to correctly propagate the information through the call graph, we once again use a version of IDFA.

Figure 3.6: Call Graph Analysis Pseudo-code

```
Graph G = FunctionCallGraph  // edge A -> B denotes A calls B
do {
  CHANGE = False
  for each function F in G
  {
     twrites = F.writes
     treads = F.reads
     tcalls = F.calls
     for each function F2 in tcalls
     {
        treads = Union(treads,F2.reads),
        twrites = Union(twrites,F2.writes)
        tcalls = Union(calls,F2.calls)
     }
     if(treads!=F.reads || twrites!=F.writes
     || tcalls!=F.calls)
     {
        CHANGE = True
        F.reads = treads, F.writes = twrites, F.calls = tcalls
     }
  }
}while(CHANGE)
```

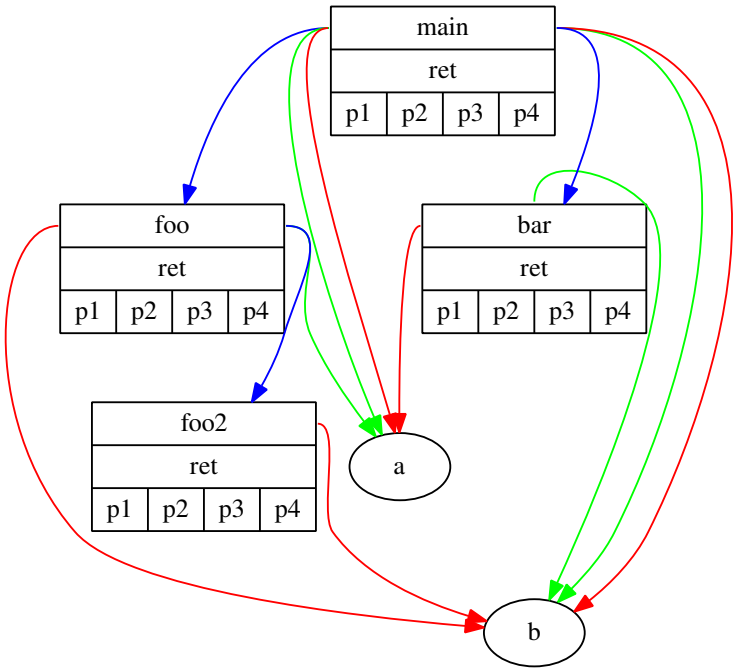

Figure 3.7: Call Graph Analysis Result

Figure 3.4 illustrates the propagation of the reads and writes through function calls. At then end of the analysis main will point to all memory locations since every function is a descendant of main in the call graph.

CHAPTER 4

DISCUSSION

4.1 Results

DDD has been implemented and is working on single file test files. However, the final results of the project have not been gathered. The next few steps involve testing the analysis on multiple files and the C-Standard library. Furthermore, the analysis will be run with the SPEC 2000 benchmarks.

4.2 Future Work

Once the project is completed with the current constraint methods, we plan on adding more constraint collection methods in order to support loads and stores of pointers, which are the last two constraints referenced by Andersen Style Pointer Analysis. If we can detect that a global variable's address is never stored to memory, then a load would not include that global variable in the points to set. Thus we would be making more precise calculations about the points-to sets. There are also some concerns about the scalability of the analysis, so we will explore methods for possible speed up of both the compile time analysis and call graph analysis.

4.3 Conclusion

Multi-core processors have significantly increased parallel computing power. Parallel computation of independent programs can of course be done without hesitation, but in order for parallel computation of a single program to be undertaken, one must take into account data dependencies. This project was made with the intention of exploring possible function level parallelism. At this point, we cannot conclude information about possible speed ups, but the results from the small test cases, are very promising.

BIBLIOGRAPHY

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, March 1976.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. *SIGPLAN Not.*, 23(7):329–338, June 1988.

[4] Sean Rul Hans Vandierendonck Koen De Bosschere. Function level parallelism driven by data dependencies. *ACM SIGARCH Computer Architecture News*, 35(1):55–62, 2007.

[5] Mickey R. Boyd and David B. Whalley. Graphical visualization of compiler optimizations. *J. Prog. Lang.*, 3(2), 1995.