Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers

Maryam Emami Rakesh Ghiya Laurie J. Hendren
School of Computer Science
McGill University, Montreal, Quebec
Canada H3A 2A7
{emami,ghiya,hendren}@cs.mcgill.ca

Abstract

This paper reports on the design, implementation, and empirical results of a new method for dealing with the aliasing problem in C. The method is based on approximating the points-to relationships between accessible stack locations, and can be used to generate alias pairs, or used directly for other analyses and transformations.

Our method provides context-sensitive interprocedural information based on analysis over invocation graphs that capture all calling contexts including recursive and mutually-recursive calling contexts. Furthermore, the method allows the smooth integration for handling general function pointers in C.

We illustrate the effectiveness of the method with empirical results from an implementation in the McCAT optimizing/parallelizing C compiler.

1 Introduction and Motivation

Alias and dependence analysis are fundamental components of optimizing and parallelizing compilers. Although traditionally studied in the context of Fortran or block-structured languages [1, 2, 8, 9], there has been increasing interest in providing accurate alias and side-effect analysis for C programs [7, 31]. Solving these problems for C rather than Fortran leads to many interesting and difficult problems including the treatment of the address-of operator (i.e. `&a`) which can create new pointer relationships at any program point, multi-level pointer references (i.e. `**a`) which enable the called function to modify alias relationships in the calling function, the integration of pointer analysis for stack-allocated variables and dynamically-allocated variables, and the proper interprocedural handling of recursion and function pointers.

SIGPLAN 94-6/94 Orlando, Florida USA
© 1994 ACM 0-89791-662-x/94/0006..$3.50

This paper reports on the design, implementation, and results of a new method for dealing with the aliasing problem in C. Our method, called *points-to analysis*, estimates the relationships between abstract stack locations. This method has been developed over the last two years within the framework of the McCAT optimizing/parallelizing C compiler, and is currently operational. The important facets of our approach include:

The points-to abstraction: Rather than compute sets of alias pairs, our method computes a different abstraction: the points-to relationships between stack locations. We say that a stack location `x` *points-to* stack location `y` at program point p if `x` contains the address of `y`.[1] In addition to providing a more compact abstraction as compared to alias pairs, the points-to information is well suited for immediate use by other analyses.

Unlike most approaches that compute *may* aliases, our analysis computes both *possible* and *definite* points-to relationships. Using the points-to abstraction, the additional overhead of providing the definite information is minimal, while at the same time providing several advantages. The first advantage is that definite points-to information can be used to sharpen the points-to analysis itself, by providing accurate killing information. For example, given the statement `*p = x`, and the information that `p` definitely points to `y`, we can kill all points-to relationships from `y`. The second major advantage is that definite relationships can be used to direct transformations like *pointer replacement*. For example, given the statement `x = *q`, and the information that `q` definitely points-to `y`, we can replace the statement `x = *q` with `x = y`. This transformation can then lead to better results in later phases of the compiler such as reducing the number of loads and stores [12].

We present the points-to abstraction and the basic flow analysis rules in Section 3 and we compare our approach to similar approaches in Section 7.

A context-sensitive interprocedural analysis:

[1] In more conventional alias analysis, the notion that `x` points-to `y` would be captured by an alias pair `(*x,y)`.

The points-to abstraction and basic flow analysis rules could be implemented in many different analysis frameworks. However, our particular approach is a structured or compositional analysis that uses *invocation graphs* to provide a context-sensitive analysis. This approach allows us to get accurate results and to correctly handle recursion.

Function pointers provide a big challenge for interprocedural analysis of C programs. In this paper we give a natural extension of the points-to analysis that gracefully includes the general problem of single and multi-level pointers to functions. In fact, as we demonstrate, the problem of instantiating function pointers in C must be done at the same time as points-to analysis in order to get a reasonably general and accurate solution.

We discuss the basic interprocedural strategy in Section 4 and then we show how to extend this to handle function pointers in Section 5. Related interprocedural approaches are compared in Sections 4 and 7.

Experimental results and applications: It is our viewpoint that any alias analysis must be implemented and tested in order to measure the actual costs and benefits of the analysis. We have completely implemented the analysis described in this paper, and in Section 6 we provide empirical evidence that our approach gives accurate and useful results. We also provide a summary of how the results of points-to analysis are used as a building block for other analyses and transformations.

Separating stack-based aliasing from array and heap-based aliasing: The problem of aliasing really comes in three varieties: (1) aliases between variable references to the stack, (2) aliases between references to dynamically-allocated storage on the heap, and (3) aliases between two references to the same array. It has become accepted that the last problem, aliases between references to the same array, requires special dependence testing methods based on analyzing the index expressions. We claim that one must also consider totally different analysis methods for stack-based aliases and heap-based aliases, and that the two problems can safely be decoupled. In the case of stack-based aliases a name exists for each stack location of interest, and one may compute some approximation of the relationships between these locations. In the case of heap-based aliases, there are no natural names for each location. In fact, one does not know statically how many locations will be allocated. Thus, a completely different approach is likely to be required. For example, Deutsch argues that a *storeless* model is more appropriate for the heap-based problems [11]. We discuss this issue further in Section 7.

2 Setting - the McCAT compiler

Our points-to analysis is implemented in the McCAT (McGill Compiler Architecture Testbed) optimizing/parallelizing C compiler. In order to provide the correct intermediate language, we designed a structured intermediate representation called SIMPLE [22, 43]. On one hand, we need to analyze real C programs (not just programs written in a toy subset of C), and therefore our SIMPLE representation faithfully represents these programs. On the other hand, we require as compact and clean a representation as possible, so that new and powerful analyses can be implemented in a straightforward and compositional fashion.

In designing the SIMPLE intermediate representation there were three major design criteria: (1) basing our analysis framework on structured (compositional) analyses, and thus using a structured intermediate representation, (2) retaining high-level variable references and type casting information, and (3) designing a compact representation for statements and expressions that includes 15 basic statements, plus explicit simplified representations of the `while`, `do`, `for`, `if`, `switch`, `break`, `continue` and `return` statements.[2] Typical simplifications include: compiling complex statements into a series of basic statements, simplifying all conditional expressions in `if` and `while` statements to simple expressions with no side-effects, simplifying procedure arguments to either constants or variable names, and moving variable initializations from declarations to statements in the body of the appropriate procedure. After simplification, points-to analysis rules need to be developed only for the 15 basic statements and the compositional control statements. The design of these rules is further simplified by the fact that each of the basic statements can have only one level of pointer indirection for a variable reference. Further details are given in [13, 22].

3 Abstract Stack Locations and Basic Analysis Rules

Traditionally, alias analysis methods have approximated aliases by sets of alias pairs. With this approximation, two variable references are said to be aliased if they refer to the same location. Typical alias pairs are of the form `(*x,y)`, `(**p,**q)`, `(*u,*v)` and so on.

3.1 Points-to Abstraction

We have chosen a different abstraction that approximates the points-to relationships between stack locations at each program point. The basic idea is to abstract the set of all accessible stack locations with a finite set of named abstract stack locations. Based on this abstraction, the approximation of interest consists of a set of points-to relationships between the abstract stack locations. For example, after the statement `p = &y`, we would say that abstract stack location `p` *points-to* abstract stack location `y`.

The key to our approach is to guarantee that each real stack location involved in a points-to relationship

[2] It should be noted that the unrestricted use of `goto` is not compositional and cannot be supported directly. Thus, our McCAT compiler provides a structuring phase that converts programs with unstructured control flow to equivalent programs with structured control flow [14].

is properly abstracted with an abstract stack location with an appropriate name. More specifically, the abstraction must obey the following two properties.

Property 3.1 *Every real stack location that is either a source or target of a pointer reference at a program point* p *is represented by* exactly *one named abstract stack location.*

Property 3.2 *Each named abstract stack location at program point* p *represents one or more real stack locations.*

An important part of our abstraction is the fact that we guarantee to provide all points-to relationships using the names of abstract stack locations that are independent of calling context. Thus, each abstract stack location corresponds to: (1) the name of a local variable, global variable or parameter; or (2) a symbolic name that corresponds to locations indirectly accessible through a parameter or global variable (of pointer type), when these locations correspond to variables not in the scope of the procedure under analysis; or (3) the symbolic name **heap**. Given that all stack locations have the appropriate names, we can define the relationships *definitely points-to* and *possibly points-to* as follows.

Definition 3.1 *Abstract stack location* `x` definitely points-to *abstract stack location* `y`, *with respect to a particular invocation context, if* `x` *and* `y` *each represent exactly one real stack location in that context, and the real stack location corresponding to* `x` *contains the address of the real stack location corresponding to* `y`. *This is denoted by the triple* `(x,y,D)`.

Definition 3.2 *Abstract stack location* `x` possibly points-to *abstract stack location* `y`, *with respect to a particular invocation context, if it is possible that one of the real stack locations corresponding to* `x` *contains the address of one of the real stack locations corresponding to* `y` *in that context. This is denoted by the triple* `(x,y,P)`.

Based on these relationships, we can define what is meant by a *safe* approximation.

Definition 3.3 *A points-to set* S *at program point* p *is a* safe *approximation if for all pairs of real stack locations* loc_i *and* loc_j, *with* `x` *as the name associated with* loc_i *and* `y` *with* loc_j:

1. *if* loc_i *points-to* loc_j *on all valid execution paths to* p, *then the points-to set* S *contains either* `(x,y,D)` *or* `(x,y,P)`.
2. *if* loc_i *points-to* loc_j *on some, but not all, execution paths to* p, *then the points-to set* S *contains* `(x,y,P)`.
3. *if* S *contains* `(x,y,D)`, *then* loc_i *must point to* loc_j *along all execution paths to program point* p.

Thus, there are two basic ways in which an approximation may not be safe: (1) a real points-to relationship is not included in S, or (2) a spurious definite points-to relationship is included in S. Of course, it is easy to find safe approximations that are not precise. For example, it would be safe to say that every abstract stack location *possibly points-to* every other abstract stack location. The goal is to find approximations that are as precise as possible. In our abstraction imprecision can be introduced by: (1) introducing spurious possible relationships, or (2) using a possible relationship in the place of a definite relationship.

3.2 L-locations and R-locations

Given that a points-to set S has been calculated for a program point p, we can define the set of abstract locations referred to by each kind of variable reference in the statement at p. L-locations are those abstract locations referred to by a variable reference on the left-hand side of an assignment statement, while R-locations are those abstract locations referred to by a variable reference on the right-hand side of an assignment statement. L-locations and R-locations are represented as pairs of the form (x, D), (x, P) where `x` is an abstract location name, and D and P indicate definite and possible locations respectively. Table 1 summarizes the L-location and R-location set for each type of variable reference allowed in the SIMPLE intermediate representation.

Note that an L-location refers to the stack location of the variable reference itself, while an R-location refers to the stack locations pointed to by the variable reference. Thus, the L-location set for `a` is simply $\{(a, D)\}$, while the R-location set is the set of all locations (x, d) such that `a` points-to `x` with the relationship of d (i.e. (a,x,d) is in the points-to set). The L-location set for `*a` is the set of stack locations pointed to by `a`, while the R-location set has one more level of indirection. That is, the R-location set includes all locations (y, d) such that `a` points-to some location `x` and `x` points-to `y`. In this case, the R-location is definite ($d = D$) only if `a` definitely points-to `x` and `x` definitely points-to `y`.

The treatment of structure references is similar, except that the field name is appended to the location names. For array references we use the notation `a[i]` to refer to an ordinary array reference, and `(*a)[i]` to refer to an array reference via a pointer. In the C source program these would both appear as `a[i]`, but in the first case `a` would have an array type, while in the second case `a` would have a pointer type. There are a variety of ways of dealing with arrays. One method is to associate an entire array with one stack location. The method presented in Table 1 uses 2 abstract stack locations for each array `a`: a_{head} is used for the location `a[0]` and a_{tail} is used for all other locations `a[1..n]`. This use of two abstract stack locations per array allows us to determine when two array pointers are aligned to

Var Ref	L-loc Set	R-loc Set
`&a`	N/A	$\{(a, D)\}$
`&a.f`	N/A	$\{(a.f, D)\}$
`&a[0]`	N/A	$\{(a_{head}, D)\}$
`&a[i]` $(i > 0)$	N/A	$\{(a_{tail}, D)\}$
`&a[i]` $(i \geq 0)$	N/A	$\{(a_{head}, P), (a_{tail}, P)\}$
`a`	$\{(a, D)\}$	$\{(x, d) \mid (a, x, d) \in S\}$
`a.f`	$\{(a.f, D)\}$	$\{(x, d) \mid (a.f, x, d) \in S\}$
`a[0]`	$\{(a_{head}, D)\}$	$\{(x, d) \mid (a_{head}, x, d) \in S\}$
`a[i]` $(i > 0)$	$\{(a_{tail}, D)\}$	$\{(x, d) \mid (a_{tail}, x, d) \in S\}$
`a[i]` $(i \geq 0)$	$\{(a_{head}, P), (a_{tail}, P)\}$	$\{(x, P) \mid (a_{head}, x, d) \in S \vee (a_{tail}, x, d) \in S\}$
`*a`	$\{(x, d) \mid (a, x, d) \in S\}$	$\{(y, d1 \wedge d2) \mid (a, x, d1) \in S \wedge (x, y, d2) \in S\}$
`(*a).f`	$\{(x.f, d) \mid (a, x, d) \in S\}$	$\{(y, d1 \wedge d2) \mid (a, x, d1) \in S \wedge (x.f, y, d2) \in S\}$
`(*a)[0]`	$\{(x, d) \mid (a_{head}, x, d) \in S\}$	$\{(y, d1 \wedge d2) \mid (a_{head}, x, d1) \in S \wedge (x, y, d2) \in S\}$
`(*a)[i]` $(i > 0)$	$\{(x, d) \mid (a_{tail}, x, d) \in S\}$	$\{(y, d1 \wedge d2) \mid (a_{tail}, x, d1) \in S \wedge (x, y, d2) \in S\}$
`(*a)[i]` $(i \geq 0)$	$\{(x, P) \mid (a_{head}, x, d) \in S \vee (a_{tail}, x, d) \in S\}$	$\{(y, P) \mid ((a_{head}, x, d1) \in S \vee (a_{tail}, x, d1) \in S) \wedge (x, y, d2) \in S\}$
`malloc()`	N/A	$\{(heap, P)\}$

Table 1: L-location and R-location sets relative to points-to set S.

the beginning of the same array. This information is useful for array dependence testing [28].

3.3 Basic Analysis Rules

The basic analysis rules are presented in Figure 1. Note that for pointer assignment statements we have a general rule that uses the L-locations for the lhs and R-locations for the rhs to compute the returned flow information. There are three basic changes to the input flow information: (1) the set of relationships killed, (2) the set of relationships that should be changed from definite to possible, and (3) the set of relationships generated. Note that any assignment between structures can be handled by breaking down the assignment into assignments between corresponding fields and then applying the basic rules.

After defining the basic assignment rule, we have defined structured or compositional rules for each of the loop and conditional constructs. We give simple versions of the **if** and **while** rules in Figure 1. The complete set of compositional rules that handle **break**, **continue** and **return** in a straightforward manner can be found elsewhere [13].

4 Interprocedural Analysis

To accurately estimate the effects of procedure calls on points-to information, we perform context-sensitive interprocedural points-to analysis. That is, when measuring the effect of a procedure call we estimate it within a specific calling context and not just summarize the information for all calling contexts. In general, a calling context depends on the chain of procedure invocations starting with **main** and ending with the particular procedure call under analysis.

The problem of ensuring that the analysis of a procedure call is specific to a particular calling context has been termed the *calling context problem* by Horwitz et al. [25], while Landi and Ryder [30] consider this to be the problem of restricting the propagation of information along realizable interprocedural execution paths. One traditional solution to this problem has been to include some context information in the abstraction being calculated [27]. Typical examples of this approach include: memory components [34], procedure strings [18, 19], assumed alias sets [30] and source alias sets with the last call-site [7].

Rather than embedding the context in the abstraction being estimated, we have chosen to follow a different strategy where we **explicitly** represent all invocation paths in an *invocation graph*. In the case of programs without recursion, the invocation graph is built by a simple depth-first traversal of the call structure of the program, starting with **main**.[3] Consider for example, the invocation graph for the program in Figure 2(a). An important characteristic of the invocation graph is that each procedure invocation chain is represented by a unique path in it, and vice versa. Using the invocation graph we can distinguish not only calls from two different call-sites of a procedure (calls to g() in Figure 2(a)), but we can also distinguish two different invocations of a procedure from the same call-site when reached along different invocation chains (call to f() in Figure 2(a)).

In the presence of recursion the exact invocation structure of the program is not known statically, and we must approximate all possible unrollings of the recursion. Figure 2(b) illustrates a program with simple recursion and the set of all possible invocation unrollings for this program. To build the graph in the case of recursion one terminates the depth-first traversal each time a function name is the same as that of one of the ancestors on the call chain from **main**. The leaf node (representing the repeated function name) is labeled as an approximate node, and its matching ancestor node

[3] The treatment of function pointers is given in Section 5.

```
/* Given a stmt S, an input points-to set, and an invocation graph node ign, return the output points-to set */
fun process_stmt (S,Input,ign) =
  if basic_stmt(S)
    return(process_basic_stmt(S,Input));
  else
    case S of
      < SEQ(S1,S2) > => return(process_stmt(S2,process_stmt(S1,Input,ign),ign));
      < IF(cond,thenS,elseS) > => return(process_if(cond,thenS,elseS,Input,ign));
      < WHILE(cond,bodyS) > => return(process_while(cond,bodyS,Input,ign));
      ...

fun process_basic_stmt(S,Input) =
  if (! is_pointer_type(S) ) /* not a pointer assignment */
    return(Input);
  else /* assignment to a pointer variable */
    kill_set = {(p,x,d) | (p,D) ∈ L-locations(lhs(S))}; /* kill all relationships of definite L-locations of lhs(S) */
    /* change from definite to possible, all relationships from possible L-locations of lhs(S) */
    change_set = {(p,x,D) | (p,P) ∈ L-locations(lhs(S)) ∧ (p,x,D) ∈ Input};
    /* Generate all possible relationships between L-locations of lhs(S) and R-locations of rhs(S).
     * The generated relationship is definite only if the L-location and R-location are both definite */
    gen_set = {(p,x,d1 ∧ d2) | (p,d1) ∈ L-locations(lhs(S)) ∧ (x,d2) ∈ R-locations(rhs(S))};
    changed_input = (Input − change_set) ∪ {(p,x,P) | (p,x,D) ∈ change_set};
    return((changed_input − kill_set ) ∪ gen_set);

fun process_if(cond,thenS,elseS,Input,ign) =
   thenOutput = process_stmt(thenS,Input,ign);
   if (elseS != {})
       elseOutput = process_stmt(elseS,Input,ign);
   else
       elseOutput = Input;
   return(Merge(thenOutput,elseOutput));

fun process_while(cond,bodyS,Input,ign) =
   /* fixed point calculation */
   do
      lastIn = Input;   Output = process_stmt(bodyS,Input,ign);   Input = Merge(Input,Output);
   while(lastIn != Input);
   return(lastIn);
```

Figure 1: Basic Analysis Rule for Points-to Analysis

is labeled as a recursive node. We indicate the pairings of these nodes with a special back-edge from the approximate node to the recursive node. It should be noted that these back-edges are used only to match the approximate node with its appropriate recursive node, and they are therefore quite different from the other tree edges which correspond to procedure calls. This scheme is completely general. Consider, for example, the invocation graph for a program with both simple and mutual recursion displayed in Figure 2(c).

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

Our overall strategy for interprocedural analysis is depicted in Figure 3, and the complete rules are given in Figure 4. The general idea is that, first, the points-to information at the call-site is mapped to prepare the points-to input set for the called procedure. This has to take into account the association of formal and actual parameters, the global variables, and the accessibility of non-local stack-locations through indirect references. Next, the body of the function is analyzed with this input points-to set and the output obtained is unmapped and returned to the call-site. Note, that by using this strategy points-to information induced by one call-site is never returned to another call-site, and similarly points-to information arriving from different call-sites is never simultaneously used to generate new points-to information. With the overall strategy being

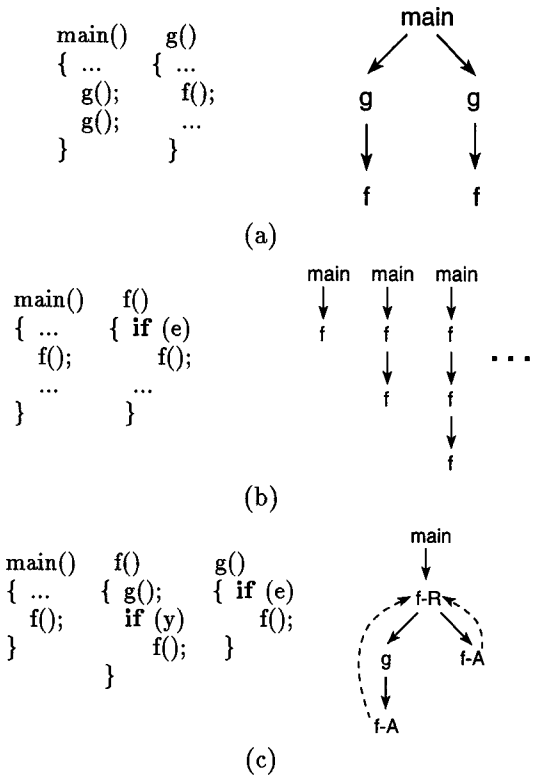

Figure 2: Invocation Contexts

clear, we now explain the strategies for mapping and unmapping points-to information, the use of invocation graph to store context-sensitive map information and the handling of recursive calls through fixed-point computations guided by the invocation graph.

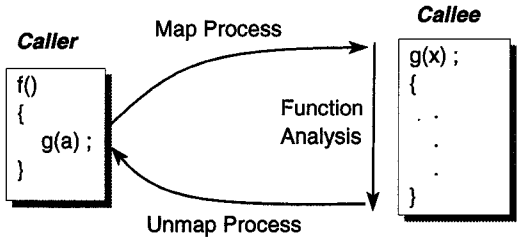

Figure 3: The Interprocedural Strategy

4.1 Mapping and Unmapping Points-to Information

When preparing the input points-to set for the called procedure from the points-to information at the call-site, the formal parameters inherit the points-to relationships from the corresponding actuals, global variables retain the same points-to relationships, while local variables are initialized to point to NULL . However, there are two important points to be considered:

• If a formal parameter or a global variable is a multi-level pointer, another pointer variable can be accessed by dereferencing it. The points-to relationships for all such indirectly accessible pointers also need to be mapped to the called procedure.

• Formals and globals can point to variables not in the scope of the procedure, which in turn can point to variables within/outside the scope of the procedure. Henceforth, we refer to variables outside the scope of the called procedure as *invisible* variables.[4]

The first problem is resolved by applying the mapping process recursively to all levels of pointer type. For the second problem, we generate special symbolic names to represent each level of indirection of pointer variables. For example, for a variable `x` with type `int**`, we would generate symbolic names `1_x` and `2_x` with types `int*` and `int`. Now, if an indirect reference, say `*x`, can lead to an invisible variable, say `b`, the corresponding symbolic name `1_x` is used to represent `b` in the points-to pairs. So a points-to pair like `(x,b,P)` at the call-site would be mapped to the pair `(x,1_x,P)`. Further, a points-to pair like `(b,c,P)`, where `c` is again an invisible variable, would be mapped as `(1_x,2_x,P)`. Simultaneously, the association of invisible variables `b` and `c` with symbolic names `1_x` and `2_x` is recorded in the invocation graph node currently under investigation, as map information. This context-sensitive information is used while unmapping and also by other interprocedural analyses [20]. Note that only the map information is context-sensitive; the symbolic names themselves are used in a context-free manner inside the procedures, by all analyses.

However, any scheme to map invisible variables to symbolic names, should take into account the following observations:

• An invisible variable should be represented by at most one symbolic name. For example if both `x` and `y` definitely point to the invisible variable `b`, it should be mapped to either `1_y` or `1_x` and not to both. Otherwise, we would have two abstract stack locations representing one real stack location, which would violate Property 3.1. So, if `b` is mapped to `1_y`, we would have the following points-to pairs: `(x,1_y,D)`, `(y,1_y,D)` and the map information would be: `(1_y,b)`, `(1_x,{})`.

• A symbolic name can represent more than one invisible variable. For example, consider the case where `x` possibly points to invisible variables `a` and `b`. Now, both `a` and `b` need to be mapped to `1_x`. Next, suppose a global variable, say `y` also definitely points to `b`. Now, either both `a` and `b` can be mapped to the symbolic name `1_x` or `a` can be mapped to `1_x` and `b` to `1_y`. However, with the first choice we would have the points-to pairs `(x,1_x,P)`, `(y,1_x,P)`, which on unmapping would generate the spurious points-to pair `(y,a,P)`, and the inaccurate pair `(y,b,P)` (instead of `(y,b,D)`). [5] So, a good mapping scheme should minimize the number of invisible variables mapped to a symbolic name to improve the accuracy of information. Our experience shows that mapping invisibles involved

[4] A similar notion of non-visible variables is given in [30].

[5] Note that the information provided is still safe, but less precise.

```
fun process_call(Input,actualList,formalList,ign,funcBody) =
  (funcInput,mapInfo)  = map_process(Input,formalList,actualList)
  case ign of
    < Ordinary > =>
      if (funcInput == ign.storedInput)  /* already computed */
        return(unmap_process(Input,ign.storedOutput,mapInfo));
      else /* compute output,  store input and output */
        funcOutput = process_stmt(funcBody,funcInput,ign);
        ign.storedInput = funcInput;     ign.storedOutput = funcOutput;
        return(unmap_process(Input,funcOutput,mapInfo));
    < Approximate > =>
      recIgn = ign.recEdge; /* get partner recursive node in inv. graph */
      if isSubsetOf(funcInput,recIgn.storedInput) /* if this input is contained in stored input, use stored ouput */
        return(unmap_process(Input,recIgn.storedOutput,mapInfo));
      else  /* put this input in the pending list, and return Bottom */
        addToPendingList(funcInput,recIgn.pendingList);
        return (Bottom);
    < Recursive > =>
      if (funcInput == ign.storedInput)  /* already computed */
        return(unmap_processs(Input,ign.storedOutput,mapInfo));
      else
        /* initial input estimate */        /* initial output estimate */
        ign.storedInput  = funcInput;      ign.storedOutput = Bottom;
        ign.pendingList  = {};               done = false;
        /* no unresolved inputs pending */
        do
          /* process the body */
          funcOutput = process_stmt(funcBody,ign.storedInput,ign);
          /* if there are unresolved inputs, merge inputs and restart */
          if (ign.pendingList != {})
            ign.storedInput = Merge(ign.storedInput,pendingListInputs);
            ign.pendingList = {};     ign.storedOutput = Bottom;
          else if isSubsetOf(funcOutput,ign.storedOutput) /* check if the new output is included in old output */
             done = true;
          else /* merge outputs and try again */
            ign.storedOutput = Merge(ign.storedOutput,funcOutput);
        while (not done);
        ign.storedInput = funcInput; /* reset stored input to initial input for future memoization */
        return(unmap_process(Input,ign.storedOutput,mapInfo)); /* return the fixed-point after unmapping */
```

Figure 4: Compositional Interprocedural Rules for Points-to Analysis

in definite relationships before the ones involved in possible relationships, gives more accurate mapping information.

Once the function is analyzed with the mapped input, the output points-to set of the function needs to be mapped back to obtain the output points-to information at the call-site. The unmap algorithm essentially consists of mapping the points-to information of symbolic names to that of invisible variables represented by them, besides that of globals. Complete details of our map and unmap algorithms are described in [13].

4.2 Recursive Procedure Calls

The cases of approximate and recursive procedure calls shown in Figure 4 work together to implement a safe and accurate fixed-point computation for recursion. As we have explained previously, all possible unrollings for call-chains involving recursion are approximated by introducing matched pairs of recursive and approximate nodes in the invocation graph. Each recursive node marks a place where a fixed-point computation must be performed, while each approximate node marks a place where the current stored approximation for the function should be used (instead of evaluating the call, the stored output is used directly).

At each recursive node we store an input, an output, and a list of pending inputs. The input and output pairs can be thought of as approximating the effect of the call associated with the recursive function (let us call it `f`). The fixed-point computation generalizes the stored input until it finds an input that summarizes all invocations of `f` in any unrolled call tree starting at the recursive node for `f`. Similarly, the output is general-

ized to find a summary for the output for any unrolling of the call tree starting in the recursive node for `f`. The generalizations of the input and output may alternate, with a new generalization of the output causing the input to change.

Let us consider the rule for the approximate node in Figure 4. In this case, the current input is compared to the stored input of the matching recursive node. If the current input is contained in the stored input, then we use the stored output as the result. Otherwise, the result is not yet known for this input, so the input is put on the pending list, and BOTTOM is returned as the result. Note that an approximate node never evaluates the body of a function, it either uses the stored result, or returns BOTTOM.

Now consider the recursive rule. In this case we have an iteration that only terminates when the input is sufficiently generalized (the pending list of inputs is empty) and the output is sufficiently generalized (the result of evaluating the call doesn't add any new information to the stored output).

5 Handling Function Pointers

In the presence of function pointers, the invocation graph cannot be constructed by a simple textual pass over the program, because a function pointer call-site cannot be bound to a unique function at compile time. A set of functions can be invoked from such a call-site, depending on the address contained in the function pointer when program execution reaches that point. Thus, proper handling of a function pointer call requires a precise estimate of this set. The simplest safe approximation for this set is the set of all functions in the program. Another possible strategy is to collect the set of all functions which have had their addresses taken, and use this set to instantiate each function pointer reference. The number and types of parameters passed cannot be safely used to refine this set, as C permits passing variable number of arguments to functions, and type casting. Either of the above methods is likely to be overly conservative and can substantially reduce the quality of flow information being collected, even if there is only one indirect call in the program. Further, these simple strategies could incur considerable cost, as each function has to be analyzed in the context of the call.

A more precise estimate can be obtained by observing the fact that the set of functions invocable from a function pointer call-site is identical to the set of functions that the function pointer can point to at that program point. To compute the points-to set of the function pointer, we need to perform points-to analysis. Points-to analysis itself needs the invocation graph of the program, as it is a context-sensitive interprocedural analysis. How do we get the invocation graph for points-to analysis? The solution lies in constructing the invocation graph while performing points-to analysis, as described below.

First, we build the invocation graph of the program following the strategy described in section 4, leaving it incomplete at the points a function pointer call is encountered. Next, we perform points-to analysis using this incomplete invocation graph. On encountering an indirect call through a function pointer, we find all the functions it can point to, according to the current points-to information. The invocation graph is updated to indicate that the indirect call can lead to invocation of any of these functions. Simultaneously, each pointed to function is analyzed in the context of the call. When analyzing an invocable function, say `f`, we consider the function pointer to be definitely pointing to `f`, as this would be the case whenever execution reaches function `f` from the given indirect call-site. Finally, the output points-to information for the indirect call is obtained by merging the output points-to sets obtained by analyzing each of the invocable functions. A more formal description of the algorithm is provided in Figure 5. The detailed description is given in [15].

```
fun process_call_indirect(Input,actualList,ign) =
    /* Get the function pointer used to make the
       indirect call */
    fptr = getFnPtr(ign)
    /* Get the set of functions pointed–to by
       fptr from current points–to information */
    pointedToFns = pointsToSetOf(Input,fptr)
    /* Initialize output of the indirect call */
    callOutput = {}
    foreach fn in pointedToFns
        /* Indicate function fn  to be
           invocable by the indirect call */
        updateInvocGraph(ign,fn)
        /* Get Invocation Graph node for fn */
        igNode = getIgNode(fn)
        /* make fptr definitely point to fn */
        igNodeInput = makeDefinitePointsTo(Input,
                                           fptr,fn)
        /* Get output for each invocable function */
        igNodeOutput = process_call(igNodeInput,
                       actualList,fn.formalList,
                       igNode,fn.funcBody)
        /* Merge all outputs */
        callOutput = Merge(callOutput,igNodeOutput)
    return(callOutput)
```

Figure 5: Algorithm for Handling Function Pointers

It should be noted that this algorithm does not add any extra cost to the analysis phase of the compiler. It simply extends the points-to analysis by using the points-to information available at indirect call-sites at the appropriate time.

We provide an example to demonstrate how this algorithm works. Consider the program given in Figure 6. Its initial incomplete invocation graph is shown in Figure 7(a). During points-to analysis, when the function pointer call `fp()` is encountered at program point A, the points-to set

```
int a,b,c;
int *pa,*pb,*pc;
int (*fp)();
main()                foo()              bar()
{ ...                 { ...              { ...
  pc = &c;              pa = &a;           pb = &b;
  if (cond)             if (cond)          /* Point D */
    fp = foo;               fp();        }
  else                /* Point C */
    fp = bar;         }
  /* Point A */
  fp();
  /* Point B */
}
```

A: (fp,foo,P) (fp,bar,P) (pc,c,D)
B: (fp,foo,P) (fp,bar,P) (pc,c,D) (pa,a,P) (pb,b,P)
C: (fp,foo,D) (pc,c,D) (pa,a,D)
D: (fp,bar,D) (pc,c,D) (pb,b,D)

Figure 6: Example Program with Function Pointers

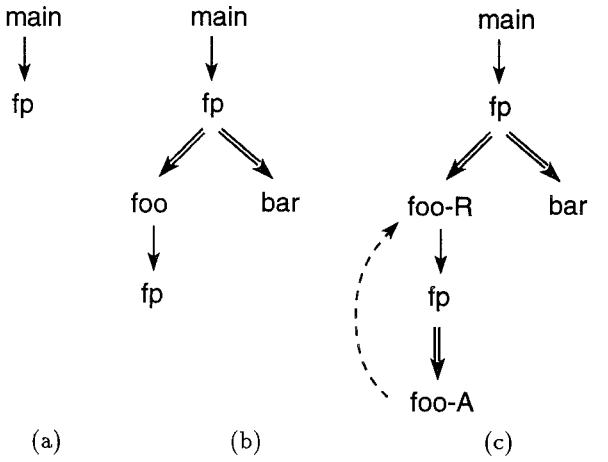

Figure 7: Invocation Graph Construction with Function Pointers

of `fp` is {`(fp,foo,P),(fp,bar,P)`}. The invocation graph is updated accordingly, as shown in Figure 7(b).[6] Next, function `foo` is analyzed with the input points-to set as {`(fp,foo,D),(pc,c,D)`} and function `bar` with the input points-to set {`(fp,bar,D),(pc,c,D)`}. Note that this set is not {`(fp,foo,P),(fp,bar,P),(pc,c,D)`} for both the functions, for reasons mentioned in the above paragraph. While analyzing the function `foo`, another function pointer call `fp()` is encountered. The invocation graph is again updated according to the current points-to set of `fp` :{`(fp,foo,D)`}, and consequently the potential call to `foo` in `main` (which is currently being considered) becomes a recursive call and is handled specially as explained in the previous section. Finally, the points-to information at program point B is obtained by merging the output points-to sets from the two potential function calls. The final points-to sets at important program points are given at the bottom of Figure 6. The final invocation graph is shown in Figure 7(c).

6 Experimental Results and Applications of Points-to Analysis

In this section we present our experimental results obtained by analyzing a set of 17 C programs. Table 2 summarizes the following characteristics of each benchmark: source lines including comments, number of statements in the SIMPLE intermediate representation, and the minimum and maximum number of variables in the abstract stacks of its functions (including symbolic variables, and all the fields of structures relevant to points-to analysis).

Our empirical results are given in tables 3, 4, 5, and 6. These results are based on our implementation in the McCAT compiler. For pointer arithmetic we assume that pointers to arrays do not cross array boundaries. For non-array pointer arithmetic our analysis supports a flag that indicates that either: (1) the pointer target stays within the presently pointed-to data structure, or (2) the pointer target can be any memory location. For the first case a warning message is issued so that the programmer can determine if this is a safe assumption. Our experiments were done with this setting.

The accuracy of points-to information collected, is best reflected in how precisely it helps in resolving indirect references in programs. We present data on points-to characteristics of indirect references in table 3. Columns 2 to 6 give the number of indirect references with the dereferenced pointer definitely pointing to a single stack location, possibly pointing to a single stack location (the other being NULL), and then to two, three and four or more stack locations. The next two columns give the total number of indirect references in the program, and the number of indirect references that can be replaced by a direct reference, by using definite points-to information. [7] Columns 9 and 10 give the number of points-to pairs used by indirect references, with the pointer target being on the stack and in the heap respectively. The *Tot* column gives their sum. The last column gives the average number of points-to pairs used in resolving an indirect reference in the program, which is the same as the average number of stack locations pointed to by the dereferenced pointer. For each multiple entry column, the first entry provides statistics for indirect references of the form `*x` and `(*x).y.z`, and the second for indirect references of the form `x[i][j]`, where `x` is a pointer to an array. Note that in our analysis we initialize all pointers to NULL. Since this initialization is not necessarily done by the user, points-to relationships contributed by it are not counted in the statistics.

[6] The double-lined edges in the figure are used just for clarity of presentation. They are not treated differently from other edges in the graph.

[7] Note that this replacement cannot be done when the dereferenced pointer definitely points-to an invisible variable.

Benchmark	Lines	#of stmts in SIMPLE	Min #of var	Max #of var	Description
genetic	506	479	33	61	Implementation of a genetic algorithm for sorting.
dry	826	212	21	43	Dhrystone benchmark.
clinpack	1231	920	11	109	The C version of Linpack.
config	2279	4549	19	188	Checks all the features of the C-language.
toplev	1637	1096	92	164	The top level of GNU C compiler.
compress	1923	1342	41	186	UNIX utility program.
mway	700	869	51	125	A unified version of the best algorithms for m-way partitioning.
hash	256	110	15	30	An implementation of a hash table.
misr	276	235	10	43	This program creates two MISR's. Their values are compared to see if the introduced errors have cancelled themselves.
xref	146	140	26	61	A cross-reference program to build a tree of items.
stanford	885	889	31	67	Stanford baby benchmark.
fixoutput	400	391	17	31	A simple translator.
sim	1422	1768	99	137	Finds local similarities with affine weights.
travel	862	543	28	55	Implements Traveling Salesman Problem with greedy heuristics.
csuite	872	781	34	55	Part of test suite for Vectorizing C compilers.
msc	148	226	20	73	Calulates the min spanning circle of a set of n points in the plane.
lws	2239	6671	64	527	Implements dynamic simulation of flexible water molecule.

Table 2: Characteristics of Benchmark Programs

Bench-mark	1 D rel		1 P rel		2 P rel		3 P rel		≥ 4 P rel		ind refs	Scalar Rep	To Stack	To Heap	Tot	Avg
genetic	2	5	14	27	0	2	0	2	2	0	54	7	38	30	68	1.26
dry	2	11	37	0	8	0	0	0	0	0	58	9	21	45	66	1.14
clinpack	7	98	0	2	0	39	0	4	0	0	150	101	197	0	197	1.31
config	8	3	34	0	0	0	0	0	0	0	45	3	45	0	45	1.00
toplev	5	0	110	0	0	0	0	0	2	0	117	5	171	0	171	1.46
compress	0	0	40	10	0	0	0	0	0	0	50	0	43	7	50	1.00
mway	31	38	0	0	0	5	0	0	0	0	74	0	79	0	79	1.07
hash	2	0	12	0	0	0	0	0	0	0	14	0	7	7	14	1.00
misr	1	3	8	0	27	0	0	0	0	0	39	0	31	35	66	1.69
xref	0	0	20	2	9	0	0	0	0	0	31	0	9	31	40	1.29
stanford	6	61	74	0	0	2	0	0	0	0	143	51	119	26	145	1.01
fixoutput	5	0	1	2	0	0	0	0	0	0	8	5	5	3	8	1.00
sim	0	0	122	231	0	0	0	0	0	0	353	0	34	319	353	1.00
travel	0	20	3	0	32	17	4	1	0	0	77	20	125	11	136	1.77
csuite	8	13	36	9	0	0	0	0	0	0	66	21	64	2	66	1.00
msc	6	0	35	0	0	0	0	0	0	0	41	6	6	35	41	1.00
lws	77	90	54	197	5	0	0	0	0	0	423	110	428	0	428	1.01

Table 3: Points-to Statistics for Indirect References

The results in this table are very encouraging. The following important observations can be made:

- The average number of stack locations pointed to by the dereferenced pointer in an indirect reference, is quite close to one for most programs, where one indicates the best possible case. The overall average is equal to 1.13, while the maximum average for a program is only 1.77. This indicates that the information collected by our points-to analysis is highly precise. The overall average is quite close to that reported by Landi et al. [31], which is 1.2.

- Overall, 28.80% of indirect references in the programs have the dereferenced pointer definitely pointing to a single stack location. Using the definite information, 19.39% of indirect references, can be replaced by direct references (when the dereferenced pointer does not point to an invisible variable). For this replacement, 67.33% of definite relationships applicable to indirect references, are useful. These are very important results, which support our strategy of collecting both possible and definite information.

- A pointer should not be pointing to NULL when being dereferenced during program execution. With this assumption, 90.76% of indirect references, have the dereferenced pointer definitely/possibly pointing to a single stack location.

- For certain benchmarks, in particular 'clinpack', 'stanford', 'sim' and 'lws', the majority of definite relationships are for indirect references of the form `x[i][j]`, (where `x` is a pointer to an array). This information is very useful for array dependence analysis, as it reduces the number of array pairs to be collected for subscript analysis [28].

- Overall, 27.92% of points-to relationships used, have heap locations as the pointer target. This underlines the need for a powerful companion heap analysis to identify disjoint accesses to heap locations [16].

- There are very few cases with three or more pos-

sible points-to relationships for an indirect reference. One of these cases (with more than 4 points-to relationships), occurs when an array of pointers is initialized (in 'toplev').

Table 4 further categorizes the points-to relationships with the pointer target on stack (column 'To Stack' of table 3), into relationships arising/directed from/to abstract stack locations representing local variables (lo), global variables (gl), formal parameters (fp) and symbolic names (sy). The statistics in the table show that most of the relationships arise from formal parameters, and are directed to symbolic names or global variables. This indicates that procedure calls generate the majority of points-to relationships, and that points-to analysis needs to be a context-sensitive interprocedural analysis to collect precise information.

Table 5 contains statistics about the total number of points-to pairs collected, obtained by summing up the number of pairs valid at each statement in the *simplified* program. Columns 2 to 4 give a classification of the points-to pairs based on their origin and target in the memory organization. The last two columns give the average and maximum number of pairs valid at a statement.

The major observation from this table, is the absence of points-to relationships from heap to locations on stack, implying that pointers in heap objects only point to other heap objects, for the given benchmark set. This supports our strategy of separating stack and heap points-to analyses. However, we need to analyze a larger set of benchmarks to further strengthen this claim, and to measure the inaccuracy introduced by our approach for those benchmarks that do have pointers from the heap to the stack.

Bench-mark	From				To			
	lo	gl	fp	sy	lo	gl	fp	sy
genetic	0	0	38	0	0	38	0	0
dry	0	0	21	0	0	9	0	12
clinpack	0	0	197	0	0	193	0	4
config	3	0	42	0	3	33	0	9
toplev	0	0	171	0	0	171	0	0
compress	11	3	29	0	0	40	0	3
mway	0	0	79	0	0	5	0	74
hash	7	0	0	0	0	0	0	7
misr	23	0	8	0	0	0	0	31
xref	0	0	9	0	0	9	0	0
stanford	0	0	119	0	0	103	0	16
fixoutput	0	0	5	0	0	5	0	0
sim	15	0	19	0	0	26	0	8
travel	2	0	123	0	14	57	0	54
csuite	12	0	52	0	8	56	0	0
msc	0	0	6	0	0	6	0	0
lws	0	0	428	0	1	350	0	77

Table 4: Categorization of Points-to Information Used by Indirect References

Landi and Ryder [30] also present empirical data on the total number of program-point-specific alias pairs collected. However, it is difficult to meaningfully compare their numbers with ours, because they greatly depend on the intermediate program representation. We do provide simple examples in Section 7 which illustrate when one method is superior to the other.

Bench-mark	Stack To Stack	Stack To Heap	Heap To Heap	Heap To Stack	A v g	Max per stmt
genetic	3901	1066	0	0	10	38
dry	512	883	198	0	7	24
clinpack	18987	0	0	0	20	91
config	136315	18	0	0	29	120
toplev	41539	6	0	0	37	100
compress	30502	1070	0	0	23	82
mway	16399	0	0	0	18	76
hash	577	207	34	0	7	18
misr	1314	706	9	0	8	25
xref	46	506	17	0	4	16
stanford	3137	364	7	0	3	30
fixoutput	3111	794	0	0	9	14
sim	7048	31174	1437	0	22	47
travel	3581	1174	0	0	8	42
csuite	4527	14	0	0	5	26
msc	221	907	88	0	5	22
lws	241291	0	0	0	35	366

Table 5: General Points-to Statistics

Bench-mark	ig nodes	call sites	#of fns	R	A	Avgc	Avgf
genetic	45	32	17	0	0	1.38	2.65
dry	19	17	14	0	0	1.06	1.36
clinpack	92	42	11	0	0	2.17	8.36
config	1068	493	49	0	0	2.17	21.80
toplev	53	29	18	0	0	1.80	2.94
compress	45	23	12	0	0	1.91	3.75
mway	44	42	21	0	0	1.02	2.10
hash	9	8	5	0	0	1.0	1.80
misr	8	7	5	0	0	1.0	1.60
xref	15	14	8	2	4	1.0	1.88
stanford	64	61	37	6	10	1.03	1.73
fixoutput	23	12	6	0	0	1.83	3.83
sim	120	47	15	2	8	2.53	8.00
travel	39	22	14	2	4	1.73	2.79
csuite	37	36	36	0	0	1.00	1.00
msc	6	5	5	2	2	1.00	1.00
lws	33	29	17	0	0	1.10	1.94

Table 6: Invocation Graph Statistics

Finally, Table 6 gives the following measurements for invocation graphs: nodes in the invocation graph, call-sites in the program, functions actually called in the program, recursive and approximate nodes in the invocation graph, and the average number of nodes in the invocation graph per call-site, and per called function.

The overall average for the number of invocation graph nodes per call-site is 1.45. Thus each call-site appears on an average on two call-chains for our benchmark set. This indicates that our approach of explicitly following call-chains is practical for real programs of moderate size, though it is theoretically exponential

in cost. However, to fully support this claim, we need to do further experimentation on larger benchmarks. If the size of the invocation graph becomes unreasonable on such benchmarks, we plan to reduce its size by sharing sub-trees that have the same or similar invocation contexts. This can be implemented by caching or memoizing the input and output points-to information for each function, and by recognizing when a particular input has already occurred. If the output has already been computed, then the sub-trees can be shared and the stored output can be used to continue the analysis.

To estimate the benefits of our technique to handle function pointers, we studied the benchmark 'livc', which is a collection of livermore loops. It has three global arrays of function pointers, each initialized to a set of 24 functions. There are three indirect call-sites in the program (each inside a loop), one involving each function pointer array. Each indirect call is made through a scalar local function pointer, which is first assigned the appropriate function pointer array element. Our algorithm constructs the precise invocation graph, instantiating each function pointer call with the corresponding 24 functions, giving a total of 203 nodes. The naive approach mentioned in section 5 would instantiate each indirect call with 82 functions (the program has total 82 functions), leading to an invocation graph with 619 nodes. An approach considering only the functions whose address has been taken, would still instantiate each indirect call to 72 functions, and construct an invocation graph with 589 nodes. Thus, both the approximations would yield very imprecise invocation graphs, as compared to our algorithm.

6.1 Applications of Points-to Analysis

The measure of success of an analysis like points-to or alias analysis is not just in measuring the number of pointed-to locations. One must also show how the results of such an analysis can be used as a building block for other interprocedural analyses and optimizing/parallelizing transformations. In our compiler framework, the points-to analysis provides: (1) point-specific points-to information and (2) a complete invocation graph with mapping information that encodes how one maps variables from a calling context to a called context.

The point-specific points-to information is very useful to compute read/write sets such as those used in constructing the ALPHA intermediate representation [21]. In these approaches one can directly compute the read and write sets based on the names of variables and the symbolic names used for invisibles. The points-to results are also critical to the support analyses required for dependence testing for array references [28]. In this context, points-to results are used to: increase the number of admissible loop-nests, decrease the number of array pairs that require testing, and allow the analysis of array subscripts that involve pointer variables. One example of an optimizing transformation is the use of definite points-to information to reduce the number of loads required in the low-level program representation [12]. In the context of fine-grain parallelizing transformations, we are currently studying the use of points-to information for providing more accurate dependence information for instruction scheduling.

The complete invocation graph and mapping information provides a convenient basis for implementing other interprocedural analyses such as generalized constant propagation [20], and practical heap analysis [16]. The important point here is that after points-to analysis is completed one does not need to worry about function pointers or the correspondence between invisible variables and the calling context. All of this information has been stored by the points-to analysis and need not be recalculated.

7 Related Work

7.1 Alias Analysis

The most closely related work is that of Landi and Ryder [30], and Choi et al. [7]. In the following paragraphs, we compare our approach with this and other related work, under different points of importance:

Alias Representation: Our points-to abstraction provides alias information in a more compact and informative manner than the exhaustive alias pairs used by Landi and Ryder. This abstraction is particularly suited for calculating stack-based aliasing, as each stack location can be given a compile-time name. It also eliminates the generation of extraneous alias pairs in certain cases. Consider the example in Figure 8. Figure 8(a) gives the points-to information, while Figure 8(b) gives the alias pairs computed by the Landi/Ryder may-alias algorithm. We can compare the two results by calculating the set of alias pairs implied by the points-to set using transitive closure. For this example, the spurious alias pair `(**x,z)` at program point S3, would not be generated by our method. However, for the example in Figure 9, the transitive closure of points-to pairs at S3 would generate spurious alias pair `(**a,c)`, which won't be reported by Landi and Ryder. The transitive reduction scheme proposed in [7] is similar to our points-to abstraction in this context.

```
main() {
    int **x, *y, z, w;
S1: x = &y; /* (x,y,D) */
S2: y = &z; /* (x,y,D)
             * (y,z,D) */
S3: y = &w; /* (x,y,D)
             * (y,w,D) */
}
```

(a) Original Program

```
S1: (*x,y) (**x,*y)
S2: (*x,y) (**x,*y)
    (*y,z) (**x,z)
S3: (*x,y) (**x,*y)
    (*y,w) (**x,z)
    (**x,w)
```

(b) Alias Information

Figure 8: Points-to Pairs vs. Alias Pairs

Must Aliases: The points-to abstraction enables the simultaneous calculation of both possible and def-

inite points-to relationships without any extra overhead. The empirical results presented in section 6 show the existence of a substantial number of definite points-to relationships, which forms very valuable information. Landi and Ryder give an algorithm only for calculation of may-aliases. An algorithm for calculating must-aliases is presented in [32]. However, it handles only single level pointers and considers the problem in isolation from may-alias computation.

Choi et al. give an example of how must-alias information can improve the precision of alias analysis. But they do not describe how to calculate must-aliases and how to integrate this calculation with the may-alias calculation.

Sagiv et al. [38] propose simultaneous collection of both universal and existential properties of programs, in their logic-based formulation of data flow analysis problems. In particular, they show how universal assertions can be used to improve the accuracy of existential assertions, using the pointer equality problem as an example. This is similar to our approach where the definite points-to information gives more precise killing information and reduces the number of spurious possible points-to pairs.

```
main() {
    int **a, *b, c;
    ...
    if(c)
S1:  a = &b; /* (a,b,D) */
    else
S2:  b = &c; /* (b,c,D) */
S3: /* (a,b,P) (b,c,P) */
}
```

(a) Original Program

```
S1: (*a,b) (**a,*b)
S2: (*b,c)
S3: (*a,b) (**a,*b)
    (*b,c)
```

(b) Alias Information

Figure 9: Points-to Pairs vs. Alias Pairs

Interprocedural Analysis: Landi and Ryder propose a *conditional* approach for context-sensitive interprocedural alias analysis. They associate an *assumed-alias* pair with every alias pair, where the validity of the alias pair at a program point is *conditional* on the validity of the assumed-alias pair at the entry node of the procedure containing the program point. They recover the calling context by determining the call-sites which can propagate the assumed-alias pair to the entry node of the procedure under analysis. Their scheme is precise for single-level pointers. However, in the presence of multi-level pointers, it can simultaneously use information arriving from different call-sites and also propagate information to extraneous call-sites. In contrast, as illustrated in our discussion of mapping/unmapping, our method can be imprecise even for single-level pointers. However, in some instances, our interprocedural scheme can avoid combining information arriving from different calls sites and will thus give more accurate results for multi-level pointers.

Choi et al. associate the last call-site `C` encountered with each alias pair. They use this information to recover context, and to avoid simultaneously using alias information arriving from different call-sites. However, they cannot distinguish information propagated by two different invocations of a procedure from the same call-site. Further, they cannot properly handle information propagated along call-chains of size greater than one. They also propose using a source alias set abstraction, but its role is not clear from their paper [33].

We do not introduce these approximations, as we explicitly propagate information along invocation paths in the program, using the invocation graph representation. Several other advantages of using invocation graphs are mentioned in section 4. Our empirical results also support the feasibility of this approach, though it is theoretically exponential in cost. Finally, none of the above techniques handle function pointers, which often occur in C programs. We gracefully integrate them in our points-to analysis framework, without incurring any extra overhead.

Dynamically-allocated objects: Landi and Ryder use the access paths as names for anonymous heap objects. They k-limit the access paths to have a finite set of object names, in the presence of recursive structures. Choi et al. name the heap objects by using the place in the program where they are created, as in [24]. They use procedure-strings and k-limiting of recursive structures [26] to improve their naming scheme. These names are then used in alias calculation. We differ from these approaches in that we claim that the stack and heap problems can and **should** be separated. The fact that the analyses can be separated is substantiated by our empirical evidence that heap-directed pointers do not in general, point back to the stack in real programs. Thus, we use a single location called *heap* in our abstract stack for the points-to analysis. All heap-directed pointers point to this location. We have designed a separate family of abstractions to capture meaningful relationships between these heap-directed pointers [16], based on the path-matrix model proposed in [23]. Both the original path matrix analysis [23] and the heap-based method proposed by Chase et al. [6] also assume that pointer fields in heap nodes only point to heap nodes, and do not point to variables (which are locations on stack). It is important to note that our points-to method provides a safe approximation even in the presence of pointers from the heap to the stack. However, there may be some loss of precision due to the abstraction of all heap locations with one abstract stack location. To date, our experiments show that this is not a problem, and that it is reasonable to decouple the stack and heap analyses.

7.2 Function Pointers

The problem of constructing the call graph of a program in the presence of procedure variables has been

previously studied [3, 17, 29, 37, 42, 44]. However, the above techniques cannot handle function pointers, because in C they are considered no different from data pointers. One can have function pointers of multiple level, as fields of structures, and as arrays. They can also be type-cast into data pointers and vice versa. Hence the full power of a points-to analysis is needed to correctly and precisely accommodate them in an interprocedural analysis.

An analogous problem of control flow analysis [40], has been studied in the domain of higher order languages. Here, the possibility of creating functions dynamically (for example, using `lambda` in Scheme) poses additional complexity. Different approaches to solve this problem have been proposed [10, 18, 35, 39, 40, 41]. In object-oriented languages, call graph analysis becomes non-trivial due to inheritance and function overloading. The method invoked from a call-site depends on the type of the receiver, and static type determination is used to estimate control flow. Type analysis techniques have been developed for C++ [36] and SELF [4, 5].

8 Conclusions and Further Work

In this paper we have presented a new method for computing the points-to information for stack-allocated data structures. This method uses the concept of abstract stack locations to capture all possible and definite relationships between accessible stack locations. The method provides context-sensitive interprocedural information, and it handles general function pointers in an integrated fashion. The points-to information can be used to generate traditional alias pairs, or it can be used directly for numerous other optimizations and transformations including pointer replacement and array dependence testing.

We have provided substantial empirical results that demonstrate that the method provides accurate and useful approximations. These results also show that it is safe and accurate to separate the stack-based points-to analysis from the structure-based approximations for heap-allocated objects. Furthermore, the method has been used as a foundation for a general purpose interprocedural analysis method [20].

The next steps in our work will be to add further optimizations to the method itself, and to measure the effect of accurate points-to analysis on other optimizations and transformations. We are also working on the companion analyses to approximate the heap. These analyses are based on a series of practical approximations of the relationships between directly-accessible heap-allocated nodes. These approximations vary from simple *connection matrices* that approximate the connectivity of nodes, to complete *path matrices* that give complete approximations of connectivity and paths between nodes.

9 Acknowledgments

A special thanks to Bill Landi for numerous e-mail discussions and his willingness to share with us output from his analysis and some of his benchmarks. We would also like to thank the PLDI program committee for their careful reviews and helpful comments. Last, but not the least, we thank all the people who have participated in developing the McCAT compiler.

References

[1] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*. pages 29–41, January 1979.

[2] J. M. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 119–131, January 1977.

[3] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering* 16,4, pages 483–487, April 1990.

[4] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Symposium on Programming Language Design and Implementation*, pages 146–160, June 1989.

[5] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN '90 Symposium on Programming Language Design and Implementation*, pages 150–164, June 1990.

[6] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. SIGPLAN Notices, Vol 25, No 6.

[7] J. D. Choi, M. G. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[8] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 247–258, June 1984. SIGPLAN Notices, Vol 19, No 6.

[9] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[10] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

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

[12] C. M. Donawa. The design and implementation of a structured backend for the McCAT C compiler. Master's thesis, School of Computer Science, McGill University, expected July 1994.

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

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

[15] R. Ghiya. Interprocedural analysis in the presence of function pointers. ACAPS Technical Memo 62. School of Computer Science, McGill University, December 1992.

[16] R. Ghiya. Practical techniques for heap analysis. ACAPS Technical Note 46, School of Computer Science, McGill University, May 1993.

[17] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3), pages 227–242, September 1992.

[18] W. L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation: an International Journal*, 1989. 2(3/4):179–396.

[19] W. L. Harrison III and Z. Ammarguellat. A program's eye view of Miprac. In *Conference Record of Fifth International Workshop on Languages and Compilers for Parallel Computing*, August 1992. Volume 757 of *Lecture Notes in Computer Science*, pages 512–537. Springer Verlag, 1993.

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

[21] L. J. Hendren, G. R. Gao, and V. C. Sreedhar. ALPHA: A family of structured intermediate representations for a parallelizing C compiler. ACAPS Technical Memo 49, School of Computer Science, McGill University, Nov 1992.

[22] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Conference Record of Fifth International Workshop on Languages and Compilers for Parallel Computing*, August 1992. Volume 757 of *Lecture Notes in Computer Science*, pages 406–420. Springer Verlag, 1993.

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

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

[25] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[26] N. D. Jones and S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Program Flow Analysis, Theory, and Applications*, pages 102–131. Prentice-Hall, 1981. Chapter 4.

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

[28] Justiani and L. J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In *Proceedings of 1994 International Conference on Compiler Construction.*, April 1994. Volume 749 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[29] A. Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 273–284, January 1993.

[30] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 SIGPLAN Symposium on Programming Language Design and Implementation*, pages 235–248, June 1992.

[31] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the 1993 SIGPLAN Symposium on Programmi ng Language Design and Implementation*, pages 56–67, June 1993.

[32] W. A. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, 1992.

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

[34] E. W. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*. pages 219–230, January 1981.

[35] A. Neirynck, P. Panangaden, and A. J. Demers. Effect analysis in higher-order languages. *International Journal of Parallel Programming*, 18(1):1–37, 1989.

[36] H. D. Pande and B. G. Ryder. Static type determination for C++. In *Proceedings of the Sixth Usenix C++ Technical Conference*, April 1994.

[37] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[38] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhem. A logic-based approach to data flow analysis. In *Proceedings of Second International Workshop on Programming Language Implementation and Logic Programming*, Volume 456 of *Lecture Notes in Computer Science*, pages 277–292. Springer Verlag, August 1990.

[39] P. Sestoft. Replacing function parameters by global variables. In *Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, London, September 1989. ACM Press.

[40] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 164–174, June 1988.

[41] O. Shivers. *Control Flow Analysis of Higher Order Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[42] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of the 1971 IFIPS Congress*. North Holland Publishing Co., Amsterdam, 1971, pages 56–60.

[43] B. Sridharan. An analysis framework for the McCAT compiler. Master's thesis, School of Computer Science, McGill University, September 1992.

[44] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*. pages 83–94, January 1980.