A Practical Interprocedural Pointer Analysis Framework

Ben-Chung Cheng* Wen-mei W. Hwu†

*Department of Computer Science
†Department of Electrical and Computer Engineering
The Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801
Email:{bccheng, hwu}@crhc.uiuc.edu
Tel: (217)244-1081
Fax: (217)333-5579

April 30, 1999

Abstract

In this paper we propose an efficient and effective algorithm to perform flow-insensitive interprocedural pointer analysis which can eliminate impossible paths. To make the algorithm efficient, we design a decoupled analysis framework where only the pointer information in each function is extracted in the function-level analysis phase, and processed with other function's pointer information in the interprocedural analysis phase. To be effective, our analysis can disambiguate accesses to a single field in a structure and union, for both stack- and heap-based storage. We have implemented the algorithm in the IMPACT compiler and evaluated its feasibility for the complete suite of SPECint92 and SPECint95 benchmarks.

1 Introduction

To reduce a program's running time, promoting memory objects to registers and hiding load instruction latencies are effective mechanisms employed by an ILP compiler [1]. However, for pointer intensive programs, determining the dependence among memory accesses is a hard problem since a single memory location may be accessed under different names, called the aliasing problem [2]. With the presence of subroutines, the aliasing problem gets even worse since a pointer may be defined and dereferenced in different functions. When multiple levels of pointers are supported, a pointer itself may have aliases. Unless the compiler has the ability to conduct complete memory disambiguation, conservative assumptions about memory dependence have to be made, leaving the code optimized in a dissatisfied way.

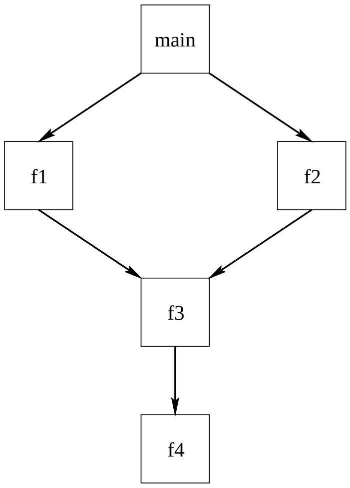
Many researchers have proposed various techniques for interprocedural pointer analysis to disambiguate memory accesses. The analysis result is represented either by alias pairs [3][4], which specify the dependence of two memory accesses, or by points-to relations [5][6][7][8], which model the storage shape graph [9]. When performing optimizations, what the compiler really needs is the alias information, which can be derived from the points-to relations [5]. Besides the above distinction, the interprocedural pointer analysis can be further classified by its flow and context sensitivity. In a flow-sensitive analysis the relative orders of statements are considered, which provides the disambiguation information at every program point [3][4][5][6]. On the other hand, a flow-insensitive analysis generates a single result containing all possible aliases for the whole program [7][10][11]. In a context-sensitive analysis, different calling contexts to a function are separated so that the alias information binded from one context will not be leaked into another one [3][4][5][6][12]. This is called the impossible paths problem. Furthermore, alias information binded from different contexts can be complete differentiated for individual pointers. Apparently, under additional cost the flow- and context-sensitive analysis can provide better results when pointers are redefined often. However, either because some algorithms can only handle a subset of the C language constructs, or because some algorithms suffer from their high complexity, no work in this field has demonstrated the capability of handling the complete suite of SPECint benchmarks, which play a pivotal role in judging the value of modern computer system designs.

To close the gap between pointer analysis algorithms and ILP compilers, in this paper we present a flow-insensitive algorithm which can disambiguate memory accesses for large C programs effectively and efficiently. Our method balances between resource requirements and analysis quality. To be efficient, the algorithm analyzes each function's body once for all calling contexts by decomposing the analysis into two phases. In the first phase, each function is analyzed individually to resolve pointers defined locally. At the mean time, pointers that can be accessed interproce-

```
int g1, g2, g3;
f1()
{
      int *pf1;
S1:   pf1 = &g1;
S2:   f3(&pf1);
S3:   *pf1 = 0;
S4:   g3 = g2 + 1;
}

f2()
{
      int *pf2;
S5:   pf2 = &g2;
S6:   f3(&pf2);
S7:   *pf2 = 0;
S8:   g3 = g2 + 1;
}
```

```
f3(int **ppf3)
{
S9:   f4(ppf3);
S10:  **ppf3 = 100;
S11:  g3 = g1 + g2;
}

f4(int **ppf4)
{
      if (....)
S12:      *ppf4 = &g3;
}

main()
{
S13:  f1();
S14:  f2();
}
```

(a)



(b)

Figure 1: Example for interprocedural pointer analysis: (a) code, (b) call graph.

durally are identified for further process. In the second phase, these interprocedurally accessible pointers exchange information through the call graph iteratively via a well-defined interface until a fixed point is reached. To be effective, the algorithm is designed to disambiguate both stack- and heap-based storage equally well. However, the proposed analysis does not differentiate individual elements in an array, nor individual elements in a linked list. Although the proposed analysis cannot provide context-sensitive information for each individual pointer in each function, it can provides context-sensitive side-effects at function levels, thus prevent impossible paths from happening. The proposed interprocedural pointer analysis has been implemented in the IMPACT compiler [13] and tested on the complete suite of SPECint92 and SPECint95 benchmarks, together with benchmarks used by other researchers.

The rest of this paper is organized as follows. In Section 2, we discuss issues of the function-level pointer analysis. In Section 3, we demonstrate how interprocedural pointer analysis proceeds. The empirical results are presented in Section 4. A comprehensive study of related work can be found in Section 5, and conclusions and future work are given in Section 6.

Production		
Access_Name	=	`Variable_Name` | `Variable_Name` *Field_Segment* | *Access_Name Dereference_Segment* | *Access_Name Dereference_Segment Field_Segment*
Dereference_Segment	=	`*`
Field_Segment	=	`.so_eo`

Table 1: Grammar of the access name.

2 Function-Level Points-to Analysis

In *C* programs, accessible objects in each function can be classified as local variables, global variables, dynamically allocated objects, and invisible objects. An invisible object in one function belongs to one of the previous four types in another function, with address being taken and passed. In terms of access methods, only local and global variables can be directly accessed, while all four classes of objects can be accessed through pointers. So the subject of conducting pointer analysis is to identify objects whose addresses are taken, and identify pointers which receive the objects' addresses. When the analysis is extended to be interprocedural, the implicit assignments between actual/formal parameters and return values/recipients are considered so that invisible objects can be disambiguated.

Consider the code example provided in Figure 1a, with the call graph depicted in Figure 1b. Statement `S12` in function `f4` modifies an invisible object, which is aliased with either `pf1` in `f1` or `pf2` in `f2`. Together with the effects of assignments `S1` and `S5`, `pf1` can point to either `g1` or `g3`, and `pf2` can point to either `g2` or `g3`. When the program is being optimized by the compiler, the load of `g2` in `S4` can be scheduled before the store of `*pf1` in `S3`. But the load of `g2` in `S8` cannot be executed before the store of `*pf2` in `S7`. And the loads of `g1` and `g2` in `S11` cannot be executed before the store of `**ppf3`, since pointer `*ppf3` may contain the addresses of `g1` or `g2`. Without pointer analysis, all these memory operations have to be executed in order.

Instead of virtually inlining all acyclic function calls to identify the possible targets of pointers, the decomposed analysis proposed in this paper can resolve them in a more economical way. In the function-level phase, pointers defined locally are identified, and invisible objects represented by *access names* will be processed in the interprocedural phase.

2.1 Points-to relation

A points-to relation is a binary relation of the form `(p, t)` where `p` is a pointer and `t` is the target [5]. Points-to relations are created because of pointer assignments, where the pointers accessed from the left-hand-side expression point to the targets pointed by the pointers accessed by right-hand-side expression. For example, `S1` in Figure 1a generates `(pf1, g1)`. When a set of points-to relations are considered collectively, they are called a *points-to graph*, or G.

2.2 Access name

For directly accessed local and global variables, the accessed region in the memory can be exactly represented by their associated variable names. But for heap-based objects and invisible objects, the only information leading to their actual location is how they are accessed. Therefore the sequence of memory operations used to access the object is encoded as the destination region's name, called the *access name*. As Table 1 shows, an access name starts from a variable name, followed by a sequence of *dereference* or *field* segments. The dereference segment is added for operators `'->'` and `'*'`, and for operator `'[]'` if the array name is a formal parameter. With field segments, individual fields in a structure/union can be differentiated. The usage of the *starting offset* (`so`) and *ending offset* (`eo`) can resolve the aliases caused by unions [6] and type casts. Using the access name notation, the points-to relation added for `S12` involving the invisible object is `(ppf4*, g3)`. The points-to graph and access names can be used together to disambiguate invisible objects, as will be explained later.

2.3 Interface variable

The binding of an invisible object can be traced based on the access name and the one-by-one mapping between the actual and formal parameters, and the function return value and recipient. In this paper, we ease the mapping process by introducing four families of systematically defined *interface variables* to replace the original function parameters and return values so that each interface variable explicitly specifies its class, rank, and origin as explained below.

- `f_`i`_`foo: the ith formal parameter of function foo.
- `a_`i`_`foo`_`bar`_`c: the ith actual parameter passed to function foo from function bar at the the cth call site.
- `or_`foo: the return value of function foo.
- `ir_`foo`_`bar`_`c: the return value recipient of function foo's return value at the cth call site called from function bar.

Interface variables are inserted to each function to participate in the points-to representation by extra assignment statements, which assign the interface formal parameter (`f_XX`) to the original formal parameter, assign the original actual parameter to the interface actual parameter (`a_XX`), assign the original function return value to the interface outgoing function return value (`or_XX`), and assign the interface incoming function return value (`ir_XX`) to the original return value recipient. The class, rank, and origin fields guarantee the uniqueness for each interface variable across the whole program.

2.4 Algorithm for the function-level analysis

Once these assignments involving interface variables are in place, access names of invisible objects will be represented by interface variables. Figure 2 lists the pseudo code for the algorithm of function level pointer analysis. It starts by inserting the interface variables into a function. Then

```
Construct_Function-Level_Points-to_Graph(function)
{
    insert interface variables
    do {
        for (each statement LHS = RHS)
            Add_Points-to_Relations(LHS, RHS)
    } while (new points-to relation is added)
}

Add_Points-to_Relations(LHS, RHS)
{
    lhs_name = Access_Name(LHS)
    rhs_name = Access_Name(*RHS)
    lhs_objects = Find_And_Create_Objects(lhs_name)
    rhs_objects = Find_And_Create_Objects(rhs_name)
    for (each object obj_l in lhs_objects) {
        for (each object obj_r in rhs_objects) {
            add (obj_l, obj_r) to the points-to graph G
        }
    }
}
```

```
Find_And_Create_Objects(access_name)
{
    Break access_name into segments by the grammar
    Use the variable name in the first segment to find the
    corresponding object obj_init
    Base_S = obj_init, Next_S = ∅
    Enqueue the rest segments into queue Q
    while (Q is not empty) {
        segment = dequeue(Q)
        for (each obj_base in Base_S) {
            Assume the name of obj_base is N
            switch (segment) {
                case "*":
                    if (obj_base's points-to list is empty)
                        Create object N* and add (N, N*)
                    Next_S = Next_S ∪ {T|∃(N,T)}
                    break
                case ".so_eo":
                    If object N.so_eo does not exist, create it
                    Next_S = Next_S ∪ N.so_eo
            }
        }
        Base_S = Next_S, Next_S = ∅
    }
    return Base_S
}
```

Figure 2: Algorithm of constructing the function-level points-to graph.

it analyzes each pointer assignment of the form `LHS = RHS` where `LHS` and `RHS` are arbitrary C expressions. The analysis is done iteratively to discover new points-to relations until a fixed point is reached. Access names for `LHS` and `*RHS` are constructed first [1], then both access names are used to search existing points-to relations. The search process initiates by considering each access name as a series of tokens of types *var*, *dereference*, and *field*, as defined by the grammar. These tokens are consumed from left to right starting from the *var* node in the points-to graph. Given node `N` in the current search result set and the next token being of *dereference* type, the search will advance to `T` if `(N, T)` $\in G$, otherwise node `N*` is added to G because `N` is a leaf node. Similarly, given node `N` in the current search result set and the next token being of *field* type, node `N.so_eo` is created, if not already existed, and included into the search result. Then bipartite points-to relations are added between the pointer set and the target set.

Table 2 lists the points-to graphs of the functions in Figure 1, where the top portion contains

[1]Note that an extra dereference segment is added to RHS.

	f1	f2	f3	f4
Resolved	(pf1,g1)	(pf2,g2)		
For further process	(a_1_f3_f1_1,pf1)	(a_1_f3_f2_1,pf2)	(ppf3,f_1_f3*) (f_1_f3,f_1_f3*) (a_1_f4_f3_1,f_1_f3*) (f_1_f3*,f_1_f3**)	(ppf4,f_1_f4*) (f_1_f4,f_1_f4*) (f_1_f4*,g3)

Table 2: Points-to relations for functions f1, f2, f3, and f4.

the points-to relations that are completely resolved by local statements, and the bottom portion contains points-to relations involving invisible objects to be further processed in the interprocedural phase.

3 Interprocedural Points-to Analysis

As briefly mentioned in the previous section, the access name of a lexically invisible object specifies the path to the destination object. So the major task performed in the interprocedural phase is to combine each function's points-to relations together, and use the access name to search for the residence of lexically invisible objects.

Figure 3 shows various data-flow paths of how points-to relations can be propagated through function boundaries. Figure 3a is the call graph, with `fc` being the center function, plus two callers `fa` and `fb`, and two callees `fd` and `fe`. If a points-to relation is propagated to a function through

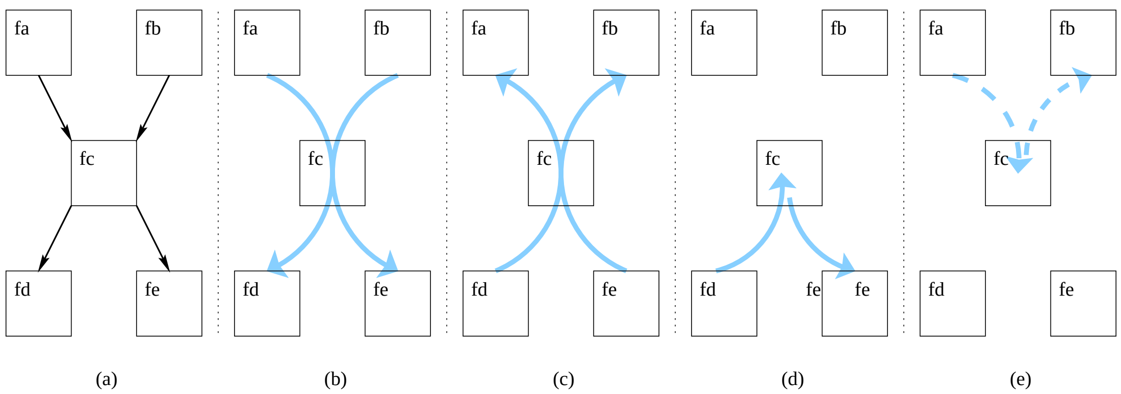


Figure 3: Propagation of points-to relations among function boundaries: (a) call graph, (b) path for inherited relations, (c) path for side-effect relations, (d) hybrid path, (e) impossible path.

```
Interprocedural_Alias_Analysis()
{
    points-to_relation_added = 1
    call-graph_changed = 1
    while (points-to_relation_added or call-graph_changed) {
        points-to_relation_added = 0
        if (call-graph_changed) {
            call-graph_changed = 0
            Rebuild the call graph from main
        }
        Traverse the call graph top-down and get inherited
            points-to relations from callers
        Traverse the call graph bottom-up and get side-effect
            points-to relations from callees

        Check all indirect call-sites for newly resolved callees
    }
}
```

```
Are_Aliases(EXP1, EXP2)
{
    acc_name1 = Access_Name(EXP1)
    acc_name2 = Access_Name(EXP2)
    objects_set1 = Find_And_Create_Objects(acc_name1)
    objects_set2 = Find_And_Create_Objects(acc_name2)
    return objects_set1 ∩ objects_set2 ≠ ∅
}
```

Figure 4: Outline of interprocedural pointer analysis.

its callers, as the arrows shown in Figure 3b, it is called an *inherited points-to relation.* Functions `fd` and `fe` can inherit points-to relations originated from both `fa` and `fb` through `fc`. On the contrary, if a points-to relation is propagated to a function through its callees, as the arrows shown in Figure 3c, it is called a *side-effect points-to relation.* Similarly, `fa` and `fb` can be affected by `fc` because of side-effect relations originated from `fd` and `fe`. Figure 3d shows the case where side-effect points-to relations reported from one callee become inherited points-to relations for another callee. Figure 3e shows the impossible path where inherited relations from one caller become side-effect relations for another caller, which can be prevented by a context-sensitive algorithm.

Figure 4 highlights the algorithm used in the interprocedural phase. We will use Figure 1 as the running example to explain it.

3.1 Inherited points-to relations

In Figure 1a, statement `S10` writes `100` to the location accessed through two levels of indirection from `ppf3`, and the access name is `f_1_f3**`. It would affect the compiler's optimization decisions if a lexically invisible object is aliased with a global object, and if two lexically invisible objects are aliases. Therefore, each intermediate pointer in an access name will be checked in all calling contexts for potential targets. For example, in the calling context created by `f1`, `f_1_f3**` is converted to a_1_f3_f1_1** first, then the first '*' leads to pf1, since a_1_f3_f1_1 points to pf1. And because (pf1, g1) exists in f1's points-to relations, (f_1_f3*, g1) is added to f3 as an inherited points-to relation. Similarly, (f_1_f3*, g2) will be added to f3 from context f2.

3.2 Side-effect points-to relations

Side-effect points-to relations are discovered by checking each function's points-to relations for pointers made of interface formal parameters or outgoing return values with non-trivial targets. A points-to relation (p, t) is non-trivial if $\texttt{t} \neq \texttt{p*}$. If the non-trivial target is not made of interface formal parameters nor global variables, it can be named trivially in the caller since the object is not pertinent in the caller's name space. For example, relation (f_1_f4*, g3) is a side-effect points-to relation in f4 since the target is not trivial. When this relation is reported to f3, the new relation added is (f_1_f3*, g3), since f_1_f4* is first converted to a_1_f4_f3_1*, and f_1_f3* is the target of a_1_f4_f3_1. Now f3 has a new side-effect relation (f_1_f3*, g3), which will be further propagated to f1 and f2 as (pf1, g3) and (pf2, g3), respectively.

3.3 Preventing impossible paths

The inherited and side-effect points-to relations discovered so far are included in Table 3a; however, spurious points-to will be added in the current configuration. Since f3 is the center function in the call graph, it receives both inherited and side-effect relations. If they cannot be differentiated, as in the current case, inherited relation (f_1_f3*, g1) will be reported to f2 as (pf2, g1), and (f_1_f3*, g2) will be reported to f1 as (pf1, g2). These spurious relations will disable some optimization opportunities, like scheduling the load of *pf1 before the store of g2 in f1 since they are now considered as aliases.

This problem can be resolved by making a slight extension to the interface variables by splitting an interface formal parameter into a *transient* formal parameter (tf_XX) and a *persistent* formal parameter (pf_XX). Each original formal parameter is assigned by both transient and persistent

	f1	f2	f3	f4
Resolved	(pf1,g1)	(pf2,g2)		
For further process	(a_1_f3_f1_1,pf1)	(a_1_f3_f2_1,pf2)	(ppf3,f_1_f3*) (f_1_f3,f_1_f3*) (a_1_f4_f3_1,f_1_f3*) (f_1_f3*,f_1_f3**)	(ppf4,f_1_f4*) (f_1_f4,f_1_f4*) (f_1_f4*,g3)
Inherited			(f_1_f3*,g1) (f_1_f3*,g2)	(f_1_f4*,g1) (f_1_f4*,g2)
Side-effect	(pf1,g3)	(pf2,g3)	(f_1_f3*,g3)	

(a)

	f1	f2	f3	f4
Resolved	(pf1,g1)	(pf2,g2)		
For further process	(a_1_f3_f1_1,pf1)	(a_1_f3_f2_1,pf2)	(ppf3,tf_1_f3*) (ppf3,pf_1_f3*) (tf_1_f3,tf_1_f3*) (pf_1_f3,pf_1_f3*) (tf_1_f3*,tf_1_f3**) (pf_1_f3*,pf_1_f3**) (a_1_f4_f3_1,pf_1_f3*) (a_1_f4_f3_1,tf_1_f3*)	(ppf4,tf_1_f4*) (ppf4,pf_1_f4*) (tf_1_f4,tf_1_f4*) (pf_1_f4,pf_1_f4*) (tf_1_f4*,tf_1_f4**) (pf_1_f4*,pf_1_f4**) (pf_1_f4*,g3)
Inherited			(tf_1_f3*,g1) (tf_1_f3*,g2)	(tf_1_f4*,g1) (tf_1_f4*,g2)
Side-effect	(pf1,g3)	(pf2,g3)	(pf_1_f3*,g3)	

(b)

Table 3: Points-to relations after performing interprocedural analysis: (a) context-insensitive analysis, (b) context-sensitive analysis with transient and persistent formal parameters.

interface formal parameters, but pointers made of transient formal variables will only carry the inherited points-to relations, and pointers made of persistent formal variables will only carry the side-effect relations. In this way, impossible paths can be prevented since inherited and side-effect points-to relations can be differentiated. The key difference between our method and other context-sensitive algorithms [3][5][6] is that instead of analyzing a function and its subsequent callees multiple times under different calling contexts, we only maintain one set of side-effect points-to relations to be propagated for all calling contexts. However, we cannot differentiate inherited points-to relations from different context-sensitive. This is the trade-off we made in order to handle larger programs. The fully resolved points-to relations employing transient and persistent interface formal parameters are shown in Table 3b.

```
typedef struct {
      int (*func)();
} s1;
typedef struct {
      s1 s;
      int *p1, *p2, *p3;
} s2;
```

```
int foo();
f5()
{
        s2 s;
S15:  s.p3 = f6(&s);
S16:  (*s.s.func)();
S17:  (*((s1 *) &s)->func)();
}
```

```
int *f6(s2 *sp)
{
S18:  sp->p1 = malloc(sizeof(int));
S19:  sp->p2 = malloc(sizeof(int));
S20:  sp->s.func = foo;
S21:  return sp->p2;
}
```

(a)

	f5	f6
Resolved		
For further	(a_1_f6_f5_1,s) (ir_f6_f5_1,ir_f6_f5_1*) (s.12_15,ir_f6_f5_1*)	(tf_1_f6,tf_1_f6*) (pf_1_f6,pf_1_f6*) (sp,tf_1_f6*) (sp,pf_1_f6*) (ir_malloc_f6_1,ir_malloc_f6_1*) (pf_1_f6*.4_7,ir_malloc_f6_1*)† (ir_malloc_f6_2,ir_malloc_f6_2*) (pf_1_f6*.8_11,ir_malloc_f6_2*)† (pf_1_f6*.0_3,foo)$^{\sharp}$ (or_f6,pf_1_f6*.8_11*)‡
inherited		
side-effect	(s.0_3,foo) (s.4_7,s.4_7*)$^{\diamond}$ (s.8_11,s.8_11*)$^{\diamond}$ (s.12_15,s.8_11*)	

(b)

Figure 5: Advanced code example with heap-allocated objects, structure fields, and function pointers: (a) code, (b) analysis results.

3.4 Dynamically allocated objects and structure fields

We have used the example in Figure 1 to demonstrate how inherited and side-effect points-to relations are propagated along the call graph. But what lacks from the intuitive example is how to disambiguate dynamically allocated objects and indirect accesses through structure fields. Therefore an advanced example is designed here to show the full strength of the proposed analysis.

In the example shown in Figure 5a, structure `s2` contains four accessible fields: a function pointer `func`, and three integer pointers `p1`, `p2`, and `p3`, where `func` is enclosed in a nested structure `s1`. One of function `f6`'s side effect is to set the function pointer field `func` with function `foo`'s address, as the points-to relation annotated by '$\sharp$' in Figure 5b. Since `func` is the first field in the `s2` structure, the starting offset of `func` is 0, and the ending offset of `func` is 3, therefore the access name for the pointer is `pf_1_f6*.0_3`[2]. Using the offsets instead of the symbolic name for structure fields eliminates the aliases caused by type casts and unions, leaving pointer assignments as the sole means for creating aliases. For example, statements `S16` and `S17` access the same function pointer field, but have different access names as `s.s.func` and `s.func` when the symbolic name is used for fields; but they are regulated into `s.0_3` if the offsets are used instead. An indirect function call can be resolved by using the call site's access name to search the points-to graph, as used in the function-level analysis. For example, once `(s.0_3, foo)` is propagated to `f5`'s points-to graph, `foo` will be included in the search result for `s.0_3*`, which is the access name for indirect calls at `S16` and `S17`. The newly resolved callee will be added to the call graph, whose points-to graph will be included and analyzed in the next iteration.

Another side-effect of `f6` is that it allocates two integer objects from the heap and stores their addresses into the `p1` and `p2` fields, as represented by the two relations annotated with '†'. It also returns the address of the second allocated object to the caller, as represented by the relation annotated with '‡'. Since the two dynamically allocated objects are not made of pertinent interface variables of the caller's name space, they can be named trivially in `f5`, as shown by the two relations annotated with '$^\diamond$'. When side-effect relation `(or_f6, pf_1_f6*.8_11*)` is reported to `f5`, it will result in relation `(s.12_15, s.8_11*)`, since `s.12_15` points to wherever `ir_f6_f5_1` points to, which points to wherever `or_f6` points to, which points to `pf_1_f6*.8_11*`, whose corresponding object in `f5` is `s.8_11*`. When the interprocedural pointer analysis finishes, if there are memory accesses involving `s.p1*`, `s.p2*` and `s.p3*` in `f5`, the compiler can perform optimizations based on the knowledge that `s.p1*` and `s.p2*` are independent while `s.p2*` and `s.p3*` are dependent.

[2] We assume the conventional sizes for generic types. But they can be assumed differently as long as their sizes are consistent for the whole program.

3.5 Behavior of library functions

Some *C* library functions have side-effects that are equivalent as pointer assignments. For example, calling "`memcpy(dest, src, 4)`" is equivalent as doing "`*(char **)dest = *(char **)src`". We can model this type of side-effects for library functions by creating a points-to graph using interface variables. For example, `memcpy`'s points-to graph is built by `(pf_1_memcpy*, pf_2_memcpy**)`. It is a safe approximation since we treat a whole array as a single object.

3.6 Disambiguating pointer targets

The top-down and bottom-up order of call graph traversal is not strictly enforced but just to reduce the number of iterations. Once the interprocedural analysis finishes, each function's points-to graph is composed of local, side-effect, and inherited points-to relations. To determine if two *C* expressions are aliases or not, their access names are constructed and processed by the *Find_And_Create_Objects* routine as shown in Figure 2. If the resultant search results are disjoint, they are definitely independent. Otherwise they are possible aliases. Due to the limit of flow insensitivity in our algorithm, the attribute of definite aliases cannot be generated without extra data-flow analysis. And the cardinality of each set will be used in Section 4 to measure the precision of the interprocedural pointer analysis algorithm.

4 Experimental Results

We have implemented our algorithm in the IMPACT compiler as part of the source-level analysis modules. In this section, we present the preliminary results of applying our algorithm on a number *C* benchmarks, including the benchmarks used by Landi and Ryder [3], and all of the SPECint92 and SPECint95 benchmarks. The experiments are setup on an HP/9000/780 workstation running at 180 MHz with 256 MB of physical memory. Our compiler modules were compiled by the HP cc compiler with the -O option.

Benchmark	Characteristics		Function-level analysis		Interprocedural analysis		Total time
	Lines	Functions	Time	Memory	Time	Memory	
clinpack	1218	11	0.17	1.86	0.01	1.61	0.18
sim	1434	14	0.39	2.06	0.05	1.68	0.44
loader	1451	29	0.34	1.81	0.14	1.89	0.48
026.compress	1503	16	0.24	1.88	0.05	1.74	0.29
129.compress	1934	24	0.13	1.77	0.03	1.65	0.16
football	2261	57	1.16	1.96	0.20	2.15	1.36
compiler	2368	40	0.27	1.77	0.03	1.80	0.30
assembler	3178	51	0.72	1.93	0.44	2.37	1.16
simulator	4508	110	0.74	1.83	0.17	2.32	0.91
130.li	7597	357	3.56	2.49	30.30	10.82	33.86
022.li	7741	357	3.61	1.91	32.81	10.83	36.42
072.sc	8639	179	2.91	1.97	1.36	3.57	4.27
023.eqntott	12053	62	0.87	1.85	0.35	2.23	1.22
008.espresso	14838	361	9.30	2.12	9.19	9.49	18.49
124.m88ksim	19092	252	3.62	2.36	1.19	4.10	4.81
134.perl	26874	276	23.65	2.31	362.47	42.10	386.12
099.go	29246	372	3.36	2.17	0.37	3.99	3.73
132.ijpeg	29290	477	11.11	2.37	126.02	22.05	137.13
147.vortex	67205	923	21.20	3.14	30.13	18.02	51.33
085.cc1	90857	1452	43.20	2.79	116.23	33.16	159.43
126.gcc	205583	2019	100.90	3.41	241.37	73.86	342.27

Table 4: Benchmark characteristics and time/size requirements.

Table 4 lists the general characteristics and time/size measurements of our algorithm for each benchmark. As shown in the first group of columns, each benchmark has at least 1,000 lines of code. The larger benchmarks have never been handled by related work: *147.vortex* (923 functions/67,205 lines), *085.cc1* (1,452 functions/90,857 lines), and *126.gcc* (2,019 functions/205,583 lines). And due to the extensive use of type casts and complex structure declarations in *132.ijpeg* and *134.perl*, they have not been handled by related work either. In the second group of columns, the average memory usage for analyzing each function and the total analysis time of the whole program are listed. The memory usage in mega bytes (MB) accounts for the internal representation of function bodies and points-to relations; and the analysis time in seconds include the process of scanning through the function body to add points-to relations for pointer assignments. The third group of columns include the total memory usage and time spent in the interprocedural analysis phase. In this phase only points-to relations of the program need to be maintained so the memory usage is reasonable. The analysis time includes the process of constructing the call graph and propagating

Benchmark	Targets per indirect access					Indirect accesses per target					Callees per indirect call				
	1	2	3	≥ 4	Avg.	1	2	3	≥ 4	Avg.	0	1	2	3	≥ 4
clinpack	18	0	0	0	1.00	18	0	0	0	1.00	0	0	0	0	0
sim	124	8	0	0	1.06	56	5	3	7	1.97	0	0	0	0	0
loader	62	2	0	0	1.03	66	0	0	0	1.00	0	0	0	0	0
026.compress	42	2	0	0	1.05	5	4	2	5	2.88	0	0	0	0	0
129.compress	27	0	0	0	1.00	15	1	2	1	1.42	0	0	0	0	0
football	61	0	1	0	1.03	61	0	1	0	1.03	0	0	0	0	0
compiler	17	0	0	0	1.00	8	0	1	1	1.70	0	0	0	0	0
assembler	108	2	0	0	1.02	104	4	0	0	1.04	0	0	0	0	0
simulator	112	2	0	0	1.02	114	1	0	0	1.01	0	0	0	0	0
130.li	613	20	8	0	1.06	443	71	20	6	1.25	0	0	0	1	3
022.li	613	20	8	0	1.06	443	71	20	6	1.25	0	0	0	1	3
072.sc	448	6	0	0	1.01	332	40	6	6	1.20	0	0	0	1	1
023.eqntott	229	7	15	0	1.15	145	27	16	7	1.48	0	2	0	2	0
008.espresso	2378	61	5	10	1.07	1881	173	42	55	1.22	0	2	3	0	2
124.m88ksim	470	39	1	3	1.11	471	27	8	4	1.11	0	0	0	0	3
134.perl	2593	62	77	128	1.51	1917	232	70	87	1.87	3	0	1	0	0
099.go	22	0	0	0	1.00	22	0	0	0	1.00	0	0	0	0	0
132.ijpeg	4317	0	0	0	1.00	3911	47	42	28	1.07	8	232	13	9	15
147.vortex	4545	231	21	3	1.06	4733	93	17	20	1.05	1	2	2	1	2
085.cc1	5217	339	92	106	1.26	4292	561	189	219	1.38	0	2	5	0	28
126.gcc	9960	946	305	148	1.34	7585	1136	536	599	1.54	0	6	10	0	49

Table 5: Accuracy of the analysis result.

the points-to relations along the call graph.

Table 4 demonstrates that our algorithm is truly a feasible solution to the interprocedural pointer analysis problem under today's computational power. Analyzing *126.gcc* consumes the most memory space due to its large number of functions. And analyzing *134.perl* takes the longest time since it contains 16 extremely intricate and large structure declarations. Each structure contains multiple pointer fields which can potentially point to all other structures, including the structure itself, causing very dense points-to relations.

In Table 5 we measure the accuracy of the points-to information generated by the proposed algorithm. When measuring the accuracy, side-effect points-to relations are considered but inherited points-to relations are ignored. This is because only one calling context is alive at a time, and the trivial points-to relations made of interface transient formal parameters can model the default context. But all inherited points-to relations need to be merged when the compiler optimizes memory instructions. The accuracy of the analysis is measured by two metrics: 1) the number of objects that an indirect memory access reaches, and 2) the number of indirect memory accesses that a object is reached by. And the general way to judge the accuracy is that the closer both metrics are to 1, the better the analysis result is. However, the standard is not absolute since pointers may be used differently in different programs. As the table shows, our analysis is accurate even though the analysis is flow insensitive.

The last group of columns in Table 5 indicate the resolution of function pointers. Resolving function pointers is a subset of resolving all pointers in a program, but tremendous benefit can be obtained since indirect call sites can be converted into direct ones thus get inlined, and the side-effects of indirect function calls can be acquired. In theory, all function pointers should be resolved if the analysis and the program is correct. But we found that *132.ijpeg*, *134.perl*, and *147.vortex* contain unresolved indirect call sites. The best explanation we can offer is that they are dead code but not removed when the benchmark was released. We used various inputs to test these benchmarks and found that these unresolved call sites are never exercised.

Due to various distinctions in generality (e.g. excluding type casts or structures/unions or function pointers), specialty (e.g. disambiguating either stack-based or heap-based storage only), and capability (e.g. size of benchmarks), it is difficult, if not impossible, to conduct a comprehensive comparison between our result with others. However, by comparing the results at the high level, we observed that the largest benchmark handled by our algorithm, *126.gcc*, is at least 3 times larger than the largest benchmark handled by other interprocedural pointer analysis algorithms [3][5][14][15]. And although not as fast as the almost-linear flow- and context-insensitive algorithm [10][14], the analysis time of our algorithm is in line with other flow-insensitive algorithms [7][11]. Most importantly, our algorithm is the first one which demonstrates the ability to resolve both data and function pointers in all SPECint benchmarks. As for the accuracy, the average numbers of targets for indirect accesses range from 1.00 to 1.51, also in line with other work [3][5][7][11][15][16].

5 Related Work

Landi et al. [3] use the may-alias representation to perform a flow- and context-sensitive analysis to compute the MOD problem. This is the pioneer work in improving the accuracy of pointer analysis, but it simplifies the analysis by ignoring *C* constructs like type casts and unions, which will affect the correctness of the result. The algorithm analyzes a program by starting from the `main` function following the interprocedural control flow graph. Since there is a specific order for functions to be analyzed in their algorithm, the bindings of lexically invisible objects are known before a function is entered. But this may cause a function to be re-analyzed multiple times. By contrast, our proposed algorithm analyzes each function once in a arbitrary order in the first phase, and only requires the points-to relations of each function to be maintained. Instead of providing complete context sensitivity, our algorithm prevents impossible paths from happening. As the experiment results show, we can analyze programs of much larger sizes as a trade-off between complexity and accuracy. Choi et al. [4] propose a sparse representation for the alias pairs, and they use the invocation path to call sites like `malloc()` to name dynamically allocated objects, again requiring the functions to be analyzed in a specific order. In our algorithm, dynamically allocated objects are represented by access names.

Emami et al. [5] propose the points-to representation to model the storage shape graph. Since their analysis is flow sensitive, their points-to relations can be classified as *definite* or *possible*. Since the bindings of lexically invisible objects are also available in advance, the syntax of pointer and target names do not contain `'*'` like ours. Since all dynamically allocated objects are modeled by a single name, these objects cannot be aggressively disambiguated. And context sensitivity is preserved in an expensive way by virtually inlining all acyclic functions, suffering from a exponential time complexity. Wilson et al. [6][15] propose the idea of *partial transfer function* to reduce the number that a function is re-analyzed. The usage of offsets for naming structure/union fields is also used in their work.

Steensgaard [10] uses type inference to collectively represent several objects in the program as a single node in the points-to representation. Because of the sharing, almost-linear time complexity is achieved, but a great deal of accuracy is also sacrificed. He extends the work to handle structure fields and type casts in [14]. The flow-insensitive analysis proposed by Anderson [11] has cubic worse-case running time, and Shapiro et. al [7] propose a hybrid method with configurable complexity between Steensgaard's and Anderson's algorithms. However, Shapiro's work only handles stack-based storage. Hasti et al. [8] propose the idea of iteratively improving the accuracy of pointer analysis in SSA form [17]. However, empirical results are not available.

Ruf [12] assesses the necessity of performing context-sensitive pointer analysis. For the benchmarks he tested, little or no precision benefit is found. What we learned from his paper is that the improvement from context-sensitive analysis may not justify the high cost if context sensitivity is obtained through virtual inlining; therefore in our algorithm partial context sensitivity can be maintained through adopting transient and persistent formal parameters, which have little effect to the analysis cost. Hind et al. [16] assess the impact of adding flow sensitivity to the analysis. The results indicate that flow-sensitive analysis provides identical precision as flow-insensitive analysis for 12 out of 21 benchmarks tested, but both are significantly better than the trivial address-taken analysis.

6 Conclusions and Future Work

This paper has described an effective flow-insensitive interprocedural pointer analysis framework which can exclude impossible paths. We have found that by decomposing the analysis into two phases, much larger benchmarks can be analyzed due to the reduced resource requirements. The capability of disambiguating memory accesses for the complete SPECint92 and SPECint95 benchmarks with resolutions down to a single field in a structure and union closes the gap between interprocedural pointer analysis algorithms and ILP compilers.

In the future work we will quantify the importance of memory disambiguation information to the code quality optimized by an aggressive ILP compiler. We will also incorporate advanced array disambiguation techniques like the Omega test [18] to improve array level dependence analysis. Since the uncertainty which prevents aggressive code optimizations can be cleared by the proposed pointer analysis, we are also looking into optimization routines like inlining indirect function calls, aggressive memory instruction scheduling, and wide-range global variable migration in the presence of function calls. The above potential optimizations will be implemented in the IMPACT compiler, and their resultant performance improvements will also be studied.

References

[1] D. M. Gallagher, *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

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

[3] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

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

[5] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242–256, June 1994.

[6] R. P. Wilson and M. S. Lam, "Effective context-sensitive pointer analysis for c programs," in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 1–12, June 1995.

[7] M. Shapiro and S. Horwitz, "Fast and accurate flow-insensitive points-to analysis," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 1–14, January 1997.

[8] R. Hasti and S. Horwitz, "Using static single assignment form to improve flow-insensitive pointer analysis," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 97–105, June 1998.

[9] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 296–310, June 1990.

[10] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 32–41, January 1996.

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

[12] E. Ruf, "Context-insensitive alias analysis reconsidered," in *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pp. 13–22, June 1995.

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[14] B. Steensgaard, "Points-to analysis by type inference in programs with structures and unions," in *Lecture Notes in Computer Science, 1060* (T. Gyimothy, ed.), pp. 136–150, Springer-Verlag, 1996. Proceedings from the International Conference on Compiler Construction.

[15] R. P. Wilson, *Efficient Context-Sensitive Pointer Analysis For C Programs.* PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA, 1997.

[16] M. Hind and A. Pioli, "Assessing the effects of flow-sensitivity on pointer alias analyses," in *Lecture Notes in Computer Science*, Springer-Verlag, 1998. Proceedings from the 5th International Static Analysis Symposium.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[18] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceedings of Supercomputing 1991*, pp. 4–13, November 1991.