Data-Flow-Based Virtual Function Resolution*

Hemant D. Pande[1] and Barbara G. Ryder[2]

[1] TRDDC, 1 Mangaldas Road, Pune, 411050, India
[2] Department of Computer Science, Rutgers University, Hill Center, Busch Campus, Piscataway, NJ 08855 USA

Abstract. Determining the type of an object to which a receiver may point at a virtual call site is crucial for compile time virtual function resolution in C^{++} programs. We show the close interdependence of *type determination* and *pointer-induced aliasing* in C^{++} and present the first data-flow-based algorithm to solve these problems simultaneously. Our polynomial-time approximation method incorporates information about program flow. Initial experiences with our prototype implementation have shown our technique to be more effective than others that use only inheritance hierarchy information. We present initial empirical results demonstrating the precision of our approach and state the intrinsic difficulty (i.e., NP-hardness) of the type determination problem for programming languages with general-purpose pointers and dynamic binding.

1 Introduction

Optimizing object-oriented programs requires dealing with dynamic binding and polymorphism, properties which differentiate object-orientation from the imperative programming paradigm. Dynamic binding prevents us from understanding the calling structure of a program by mere observation of the code; in this respect the challenge it presents to compiler writers is much like that of function pointers in C. In C^{++} general-purpose pointer usage exacerbates the difficulty of dealing with dynamic binding, because discerning object types may involve tracing aliases through pointers. Our analysis technique deals with general-purpose pointers and combines the problems of pointer-induced aliasing and type determination for C^{++}, solving them with a single data-flow-based approximation algorithm.

Contributions Previously, we presented a static type determination algorithm for C^{++} programs restricted to those using only single level pointers [PR94a]. In this context, type determination can be solved independent of aliasing. However, in the presence of multiple level pointers, these two problems cannot be precisely solved separately, but are interdependent (see Section 3). Ours is the first combined approximation algorithm for these problems. Because the combined problem is provably NP-hard (see Section 2), our polynomial-time algorithm computes an *approximation* of the solution, which has been shown to have acceptable precision in our initial empirical experiments (see Section 4). We also show the gains our analysis realizes over methods based solely on the inheritance hierarchy.

* This research was supported, in part, by funds from NSF grants GER90-23628, CCR-92-08632 and Siemens Corporate Research.

Motivating Example Discerning the type of an object to which a receiver may point at a virtual call site not only reveals the possible calling structure of the program, but also can enable many optimizations, even if the type is not uniquely determined. The example in Figure 1 illustrates the significance of virtual function resolution in practical issues of run-time efficiency and benefits to other optimizations. We assume that the boolean conditions in *if* statements are side-effect-free and hence inconsequential to the analysis.

```
class Base {
public:
        virtual void foo ( );
        virtual void bar ( );
        virtual void baz ( );
} *a, *p, *q;

void Base::foo ( ) {
   n0 : a = new Base;}

void Base::bar ( ) {...}

void Base::baz ( ) {...}
```

```
class Derived : public Base {
public:
        void foo ( );
        void bar ( );
}  r, *s;

void Derived::foo ( ) {
   n1 : a = new Derived;
   n2 : printf ("hello world\n");}

void Derived::bar ( ) {
   Derived *t;
   n3 : t = s;}
```

```
main ( )  {
    if (-) n4 : p = new Base;
    else   n5 : p = new Derived;
    n6  : s = &r;
    n7 : r.bar ( );
    n8 : q = new Base;
    n9 : q->foo ( );
    n10 : a->bar ( );
    n11 : p->baz ( );}
```

Fig. 1. Example for virtual function resolution

At statement $n8$ in Figure 1, the pointer q is made to point to an object of class *Base*, and then immediately used at statement $n9$ as the receiver for a virtual function invocation. Under these circumstances $Base :: foo()$ will be invoked on all executions notwithstanding the virtual nature of the invocation. Since the virtuality of $Base :: foo()$ is not utilized, the invocation can be compiled as a direct function call, thereby reducing the run time overhead of virtual invocation.

Limiting the scope of invocation to $Base :: foo()$ and eliminating $Derived :: foo()$ from consideration may benefit other analyses. The assignment at statement $n1$ and printing of *hello world* at $n2$ will not appear as possible side effects of the invocation at statement $n9$. Significantly, our algorithm will be able to determine that $n10$ is a call of $Base :: bar()$ and never $Derived :: bar()$, because the receiver a may only point to an object of type $Base$. Therefore, call site $n10$ can be considered non-virtual. Given the potential disparity in side effects of different implementations of a virtual function, this ability to perform unique resolution can significantly improve the precision of subsequent analyses. Note that this improvement required the ability to do type determination while accounting for interprocedural flow from $main$ to $Base :: foo()$ and back.

Resolving a virtual function invocation to a unique function call also may create possibilities for inlining, resulting in elimination of function call overhead. Inlining a function call may also provide opportunities for various intraprocedural optimizations which would have been impossible if attempted in the untransformed program.

A transformation from virtual invocation to function call is sometimes possible without unique resolution of the receiver type. For example at statement $n11$, the receiver p may point to an object of class $Base$ or $Derived$, because of the assignments at $n4$ and $n5$. Since the receiver type is not unique, a naive approach may result in retaining the invocation as virtual. However, since class $Derived$ inherits the function $baz()$ from class $Base$ without redefining it, $n11$ may still be safely compiled as a function call to $Base :: baz()$. In general, even if the receiver at the virtual invocation site does not point to a unique class, but all the receiver types in the inheritance hierarchy utilize the same implementation of a virtual function, the virtual invocation may be compiled as a direct function call. Nevertheless, such hierarchy information would fail to resolve the virtual calls at statements $n9$ and $n10$, as there are multiple implementations of $foo()$ and $bar()$ in the hierarchy. An approach which takes into account the data flow (as we saw earlier) is necessary to uniquely resolve these virtual calls.

For architectures which use deep pipelining and speculative execution, the issue of accurate control flow prediction assumes significant importance. Using type determination to replace virtual invocations with function calls, when the target function is known at compile time, would yield benefits comparable to those obtained by profile-based prediction for C^{++} [CG94]. A uniquely resolved call site would eliminate pipeline stalls, as the target of the call is unambiguously known. A related empirical study of behavioral differences between C and C^{++} programs reveals that C^{++} functions are usually small (in terms of lines of code) and C^{++} programs contain more calls than C programs [CGZ95]. The higher calling frequency usually degrades analysis performance because of the inherent difficulty of interprocedural analysis contrasted with that of traditional intraprocedural analysis. Therefore, these findings suggest that inlining a uniquely resolved function may be a crucial optimization.

Even when unique resolution is not possible, short-listing the functions may allow the compiler to replace the late binding mechanism of virtual call with appropriate function calls within a decision statement. Branch prediction techniques may then be applied to improve the execution performance of the program. Additionally, short-listing can focus further analyses on selected functions, rather than on the entire pool of functions with the same name, potentially saving analysis time and allowing building of a more precise call graph.

While unique resolution of virtual functions seems to contradict the encouraged usage of polymorphism in object-oriented languages, run-time analyses have shown that polymorphism is used selectively, leaving much room for unique resolution of non-polymorphic calls as well as optimization of polymorphic calls [CG94].

Outline Section 2 presents key concepts used in our analysis and defines the ICFG, our internal graph representation of the C^{++} program. We also state our theoretical complexity theorem and outline its proof. Section 3 introduces our approximation algorithm and compares it to the *conditional analysis* approach we previously used for C programs. Our algorithm is a flow-sensitive propagation of information on paths in the ICFG: Figure 2 of Section 3 gives a high level overview of the algorithm which follows a typical data flow worklist scheme. Section 4 includes our implementation results, starting with a description of our test suite of 19 C^{++} programs. We discuss our experiences with virtual function resolution and compare our approach with simpler approaches (see Figure 6(b)). We conclude with our future research plans. Because of space limitations, we have excluded our related work section and some of our empirical results; they are available in technical report number LCSR-TR-269, at **ftp://www.cs.rutgers.edu/pub/technical-reports**.

2 Problem Definition

Program Representation A *control flow graph* (CFG) for a function consists of nodes which represent single-entry, single-exit regions of executable code, and edges which represent possible execution branches between code regions. We represent a program with an interprocedural control flow graph (ICFG), which intuitively is the union of CFGs for the individual functions comprising the program [LR91, PR94a]. Formally, an ICFG is a triple ($\mathcal{N}$, $\mathcal{E}$, ρ) where $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of edges and ρ is the *entry* node for `main`. $\mathcal{N}$ contains a node for each simple statement in the program, an *entry* and *exit* node for each function, and a *call* and *return* node for each invocation site. An intraprocedural edge into a *call* node represents the execution flow into an invocation site, while an intraprocedural edge out of a *return* node represents control flow from an invocation site once the invoked function has returned.[3] For a non-virtual function call, we represent the control flow into the called function by an interprocedural edge from a *call* node to the corresponding *entry* node. Similarly, we represent the return of control from the called function by an interprocedural edge from the *exit* node to the *return* node. However, virtual function invocation makes it impossible to determine the correspondence between a *call* and *entry* before analysis, since the function invoked depends on the type of the receiver at the call site. Establishing the interprocedural edge(s) from a *call* node representing

[3] We use the terms *call* and *invocation* interchangeably.

a virtual function invocation to appropriate *entry* node(s) and from *exit* node(s) to the corresponding *return* node(s), (i.e., resolving virtual functions), is a major goal of the algorithm presented in this paper.

- An ICFG path is *realizable* if, whenever a called function on that path returns, it returns to the corresponding *return* node of the call site which invoked it [LR91]. Not all ICFG paths are realizable.
- A realizable path is *balanced* if for each intermediate *call* node, the path contains a corresponding *return* node representing the return of control from the called function.[4] Intuitively, the first and the last node on a balanced path belong to the same function. Moreover, they are in the same incarnation of that function since every called function on the path (perhaps recursively) must return control before the path terminates.
- *Objects* correspond to locations that can store information; *object names* provide ways to refer to them. We associate names with static memory locations and dynamic (i.e., heap) locations (created by *new*). For static storage, the name-storage association is created through a variable declaration statement. For each heap location, we create a name new_{pp} where pp is the program point representing the creation site for the location. An *object name* is a variable name or a heap location name, and a possibly empty sequence of dereferences.
- An *alias* exists at a program point when two or more object names refer to the same location as a result of program execution to that program point. We represent aliases by unordered pairs of object names (e.g. $<v, *p>$). The order is unimportant since the alias relation is symmetric.
- *Type determination* involves calculating the type of the object pointed to by a pointer at a program point as a result of an execution to that program point. We represent this information by a pair consisting of a pointer and an associated type (e.g. $<p \Rightarrow C>$), called a *pointer-type* pair.
- A realizable path from ρ is called *consistent* if, for every edge $\ll$`call`, `entry`$\gg$ on the path, where `call` represents a virtual call with receiver *rec*, the execution defined by the subpath from ρ to `call` implies a pointer-type pair $<rec \Rightarrow C>$ at `call` such that the virtual function represented by `entry` is invocable from `call`. Non-consistent paths do not correspond to any possible execution sequence.
- The *precise*[5] solution for static type determination and aliasing at a program point is a set of pointer-type and alias pairs, each of which is a result of an execution on some consistent path to that program point.

Theorem 1. *In the presence of pointers and virtual functions in C^{++}, precise program-point-specific type determination and aliasing is NP-hard.*

The theorem proof works by reducing the 3-SAT problem to type determination and aliasing [Pan96].

---

[4] We defined the terms *realizable* and *balanced* paths independently [LR91, PR94b], and found later that the ideas already existed in literature, referred to as *valid* and *complete* respectively in [SP81]. [RHS95] also addresses these issues.

[5] Under the standard assumption of static analysis that all intraprocedural paths are executable.

3 Type Determination and Aliasing Algorithm

Aliasing and Type Determination In programs restricted to single level pointers, one pointer cannot be aliased to another, as this requires multiple levels of indirection [LR91]. As a result, when a pointer changes its value (to point to an object of another type), it does not affect the value of any other pointers [PR94a]. In this context, type determination impinges on aliasing since the receiver types decide which virtual function is invoked at a call site, and the invoked function can affect aliasing. Nevertheless, aliasing plays no part in type determination.

Such a separation does not occur when we allow multiple level pointers. As an example, the ICFG node "`m : p = &q;`" creates alias $<*p, q>$. Suppose subsequently on an execution path, "`n : *p = &r;`" creates pointer-type pair $<*p \Rightarrow \mathit{type}(r)>$. In the absence of information that the alias pair $<*p, q>$ holds before node `n`, we would not be able to infer $<q \Rightarrow \mathit{type}(r)>$ at `n` and the type determination would be rendered incorrect and unsafe. Thus, aliasing affects type determination and *vice versa.*

In this section, we formulate the combined problem and state our tractable approximation of it. Then, we describe our algorithm at a high level, aided by examples.

Problem Formulation *Conditional analysis* [LR91] involves analyzing execution flow in a function, assuming that certain information holds at the entry of the function. We use this approach to define the following terms.

- A balanced path to an ICFG node `n` from entry node `e` of the function containing node `n` is called *conditionally consistent* with respect to an assumption set S of alias and pointer-type pairs, if for every edge $\ll$`call`, `entry`$\gg$ on the path, where `call` represents a virtual call with receiver *rec*, the following is true:

  Given that all the alias and pointer-type pairs in S hold at `e`, the execution defined by the subpath from `e` to `call` implies a pointer-type pair $<\mathit{rec} \Rightarrow C>$ at `call` such that the virtual function represented by `entry` is invocable from `call`.

  We denote such a path by $\mathcal{P}_{n,S}$.
- We define the *conditional type determination problem* as deciding for an ICFG node `n`, a pointer-type pair PT, and a set of alias and pointer-type pairs S, whether:

  there exists a conditionally consistent path with respect to S to node `n` on which PT holds, **and** there exists a consistent path from ρ, the entry node of `main` in the ICFG, to the entry node of the function containing `n` on which every pair in S holds.
- Similarly, we define the *conditional aliasing problem* as deciding for an ICFG node `n`, an alias AP, and a set of alias and pointer-type pairs S, whether:

  there exists a conditionally consistent path with respect to S to node `n` on which AP holds, **and** there exists a consistent path from ρ to the entry node of the function containing `n` on which every pair in S holds.

Approximation Formulation Since the above formulation is computationally intractable, we approximate the joint solution of these two related problems by considering an approximate assumption set S' to contain at most *one* alias or pointer-type from S, chosen arbitrarily.[6] We use the pair in S' (i) to approximate a conditionally consistent path to `n`, and (ii) as the only necessary assumption for conditional analysis. This approximation leads to a safe overestimate of consistent paths and conditional analysis solution [Pan96].

A major difficulty in interprocedural data flow analysis is ensuring that the propagation of information only occurs on realizable paths in the program representation. The reaching alias pair used by the Landi-Ryder pointer aliasing algorithm [LR92] acts as an approximation of the call stack at the call site; that is, when data flow information is gathered within a called procedure, the corresponding reaching alias determines to which possible call sites this information will be propagated by choosing those sites with "matching" reaching aliases. Our algorithm similarly uses the assumptions in our predicates (i.e., S') to approximate possible consistent paths in the ICFG.[7] Thus, our algorithm is *context sensitive* (although approximate), thereby analyzing called functions within different calling contexts. It is *flow sensitive* because it propagates data flow information across calls to called procedures and then along paths within those procedures and back again to the calling procedure.

We define two predicates, *points-to-type*[8] and *may-hold*, with the following interpretations.

points-to-type (n, $AAPT$, $<p \Rightarrow C>$) if there exists a consistent path from ρ to the *entry* node of the procedure containing node `n`, on which an alias or pointer-type pair $AAPT$ (if any)[9] holds **and** there exists a conditionally consistent path $\mathcal{P}_{n,\{AAPT\}}$ to `n` on which $<p \Rightarrow C>$ holds. In Figure 1, we will have a *true* valued *points-to-type* ($n3$, $<s \Rightarrow Derived>$, $<t \Rightarrow Derived>$), as $<s \Rightarrow Derived>$ holds on the path $\rho{\cdot}n4{\cdot}n6{\cdot}n7{\cdot}entry_{Derived::bar}$ **and** $<t \Rightarrow Derived>$ holds on the path $entry_{Derived::bar} \cdot n3$ assuming $<s \Rightarrow Derived>$ holds at $entry_{Derived::bar}$. Similarly, *may-hold* (n, $AAPT$, $<a, b>$) if there exists a consistent path from ρ to the *entry* node of the procedure containing node `n`, on which $AAPT$ (if any) holds **and** there exists a conditionally consistent path $\mathcal{P}_{n,\{AAPT\}}$ to `n` on which $<a, b>$ holds. In Figure 1, we will have a *true*-valued *may-hold* ($n3$, $<*s, r>$, $<*t, r>$), as $<*s, r>$ holds on the path $\rho \cdot n4 \cdot n6 \cdot n7 \cdot entry_{Derived::bar}$ **and** $<*t, r>$ holds on the path $entry_{Derived::bar} \cdot n3$ assuming $<*s, r>$ holds at $entry_{Derived::bar}$. For efficiency, we have designed our approximation algorithm so that only those *may-hold* and *points-to-type* predicates are created which

[6] The choice of one alias or pointer-type was suggested by the use of one alias pair to encapsulate calling context in the Landi-Ryder pointer aliasing algorithm [LR92]. It was shown in [LR91] that a single alias pair precisely represents calling context, if a program is restricted to only a single level of dereferencing.

[7] Our algorithm is *approximate* and *safe* in the sense that it will propagate information on all consistent paths as well as possibly some inconsistent paths.

[8] In our previous papers [PR94a, PR94b], *points-to-type* was called *points-to*.

[9] We will consistently use $AAPT$ to denote a specific "Assumed Alias or Pointer-Type" pair. $AAPT$ may be $\emptyset$.

would return *true* for the conditional aliasing and type determination problems, respectively. Having obtained a solution for these predicates, solving the aliasing and type determination problems is easy. For example, if *points-to-type* (n, $AAPT$, $<p \Rightarrow C>$) is *true*, $<p \Rightarrow C>$ is in the type determination solution at node `n`.

Algorithm Overview Our combined algorithm for aliasing and type determination is a worklist based, fixed point iteration method which is both *safe* and *approximate*: If there exists an execution path to ICFG node `n` on which a pointer p points to an object of type C, our algorithm will report *points-to-type* (n, $AAPT$, $<p \Rightarrow C>$) for some entry assumption $AAPT$. Similarly, if there exists an execution path to node `n` on which $<a, b>$ holds, our algorithm will report *may-hold* (n, $AAPT$, $<a, b>$) for some $AAPT$.

The worst case complexity of our algorithm is polynomial in the number of ICFG nodes. However, we can show that in practice the algorithm runs in time linear with respect to the size of *may-hold* and *points-to-type* solution generated by our method [Pan96].

A high level description of our algorithm appears in Figure 2. The algorithm has three main phases which are discussed using examples in Sections 3.1–3.3. Firstly, the predicates *points-to-type* and *may-hold* and the worklist are *initialized* (see Section 3.1). Secondly, we *introduce* certain *true*-valued predicates at pointer assignments (using **intra-alias-type-introduction**) and at parameter binding sites (using **inter-alias-type-introduction**) (see Section 3.1). These initial predicates are placed on the worklist. Thirdly, the algorithm performs the usual fixed point iteration, until the worklist is empty. That is, a predicate is removed from the worklist and *propagated* through successor nodes using appropriate functions determined by the node type. This propagation of information occurs in the `while` loop of Figure 2. Intraprocedural propagation is explained in Section 3.2; the interprocedural propagation functions (e.g., **alias-type-propagation-from-call**) are explained in Section 3.3. The propagation functions make additional predicates *true* and put them on the worklist. Whenever a predicate becomes *true* for the first time, it is placed on the worklist. We refer to this action as **make-true**. Once marked *true*, a predicate stays *true*. A predicate goes on the worklist at most once, guaranteeing the termination of our algorithm.

The calculation of a fixed point for *points-to-type* and *may-hold* is tantamount to the solution of a standard monotone data flow framework defined on a lattice whose elements are sets of (*assumption, alias/pointer-type*) tuples. For simplicity, we have chosen to provide an intuitive explanation of the monotone transfer functions and solution procedure. A formal treatment, including the underlying program representation, lattice definition and monotone function space appears in [Pan96].

Calculation of Approximate Assumption Sets Before providing algorithm details, we briefly describe some auxiliary functions used to capture type and aliasing effects on entry of an invoked function from the types and aliases present

```
// initialization of information (Section 3.1)
lazily set all possible predicates to false;
set worklist to empty;
// introduction of aliases and pointer-type pairs (Section 3.1)
intra-alias-type-introduction ( ); // (Figure 4)
for each non-virtual call to entry
        inter-alias-type-introduction (call, entry);
// propagation of aliases and pointer-type pairs
while worklist is not empty
        remove (M, AAPT, APT) from worklist
        if M is a call node // (Section 3.3)
            alias-type-propagation-from-call (M, AAPT, APT);
        elseif M is an exit node // (Section 3.3)
            if APT is an alias pair
                alias-implies-alias-from-exit (M, AAPT, APT);
            if APT is a pointer-type pair
                type-implies-type-from-exit (M, AAPT, APT);
        else // intraprocedural propagation (Section 3.2)
            for each N ∈ successor (M)
                if N is a pointer assignment // (Figure 5)
                    if APT is an alias pair
                        alias-implies-alias-thru-assign (N, M, AAPT, APT);
                        alias-implies-type-thru-assign (N, M, AAPT, APT);
                    if APT is a pointer-type pair
                        type-implies-type-thru-assign (N, M, AAPT, APT);
                        type-implies-alias-thru-assign (N, M, AAPT, APT);
                // propagate directly through N
                elseif APT is an alias pair
                    make-true(may-hold (N, AAPT, APT))
                else
                    make-true(points-to-type (N, AAPT, APT))
```

Fig. 2. A High level description of the algorithm

at the invocation site. These functions calculate the approximate assumption sets described earlier. The first two functions, **bind0** and **type-bind0**, are used during the introduction phase (Section 3.1) and the rest during interprocedural propagation (Section 3.3). In these descriptions, `call` and `entry` represent ICFG nodes whereas *alias* and *pointer-type* are specific pairs.

bind0(`call,entry`) : This function calculates those *aliasing effects* from `call` to `entry` requiring no information from the predecessor(s) of `call`. For example, if $\&a$ is passed as an actual to the formal f, $<*f, a>$ is created at `entry` regardless of any aliases a may have at `call`; so $<*f, a> \in$ **bind0**(`call,entry`).

type-bind0(`call,entry`) : This function calculates those *type effects* from `call` to `entry` requiring no information from the predecessor(s) of `call`. For example, suppose a is an object of class B and $\&a$ is passed as actual to the formal f; then $<f \Rightarrow B> \in$ **type-bind0**(`call`, `entry`).

```
Class B { public:                         C *s;    D *r;
  int b1;};                               int *p, *q, i;
                                          main () {
Class C { public:                           n1 : p = &i;
  int foo (int *foo1, int *foo2);           n2 : q = p;
  int bar (D *bar1; B *bar2);};             n3 : s = new C;
                                            c1 : s->foo (p,q);
Class D { public:                           n4 : r = new D;
  B *d1; };                                 n5 : r->d1 = new B;
                                            c2 : s->bar (r, r->d1);}
```

Fig. 3. Example for binding functions

bind(`call`,`entry`,*alias*) : This function represents the propagation of *alias* reaching the `call` to the corresponding `entry`. Depending on the actual-formal associations, *alias* may manifest itself at `entry` and/or may give rise to additional alias pairs. In Figure 3, $<*p, *q>$ is created at `n2` and reaches $call_{c1}$, resulting in:

$$\textbf{bind}(call_{c1}, entry_{C::foo}, <*p, *q>) = \\ \{<*foo1, *foo2>, <*foo1, *q>, <*foo2, *p>, <*p, *q>\}$$

alias-bind(`call`,`entry`,*pointer-type*) : This function calculates the *alias effects* of *pointer-type* present at `call` which fall into the following two categories: aliases between appropriate dereferences of two formals (i.e., the first pair below) and aliases between the dereference of an actual and the corresponding formal (i.e., the last pair). In Figure 3, **alias-bind**($call_{c2}, entry_{C::bar}, <r \rightarrow d1 \Rightarrow B>$) includes:

$$\{<bar1 \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1>, <r \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1>\}$$

type-bind(`call`,`entry`,*pointer-type*) : This function calculates the *type effects* of *pointer-type* present at `call` on `entry`. Depending on the actual-formal bindings at `call`, *pointer-type* may simply propagate to `entry` and/or may create a pointer-type pair involving the corresponding formal. In Figure 3, **type-bind** ($call_{c2}$, $entry_{C::bar}$, $<r \rightarrow d1 \Rightarrow B>$) includes:

$$\{<r \rightarrow d1 \Rightarrow B>, <bar2 \Rightarrow B>\}$$

3.1 Initialization and Introduction Phases

Conceptually, the algorithm starts by initializing all the *points-to-type* and *may-hold* predicates to *false*.[10] We also initialize the worklist to empty. The intraprocedural aspects of the introduction phase are summarized in Figure 4. This introduces pointer-type and alias pairs generated locally at a pointer assignment node.

for each node `n` in the ICFG
    If `n` is
        `n : p = new B;`
            make-true(*points-to-type* (n, $\emptyset$, $<p \Rightarrow B>$))
            make-true(*may-hold* (n, $\emptyset$, $<p \rightarrow mem_k, new_n.mem_k>$))†
        `n : p = &r;`
            make-true(*points-to-type* (n, $\emptyset$, $<p \Rightarrow type(r)>$))
            make-true(*may-hold* (n, $\emptyset$, $<p \rightarrow mem_k, r.mem_k>$))

†We use $<p \rightarrow mem_k, new_n.mem_k>$ to denote all aliases involving corresponding members of the class.

Fig. 4. Intraprocedural introduction phase : **intra-alias-type-introduction**

The function **inter-alias-type-introduction**(`call`,`entry`) has the following task: for each AP in **bind0**(`call`,`entry`), **make-true**(*may-hold* (`entry`, AP, AP)); and for each PT in **type-bind0**(`call`,`entry`), **make-true** (*points-to-type* (`entry`, PT, PT)).

3.2 Intraprocedural Propagation

We present the salient features of intraprocedural propagation in Figure 5, referring to appropriate functions from Figure 2. Accompanying each function we list the predicate(s) being propagated from node `m`, the pointer assignment successor `n` through which they propagate, and finally the resulting predicate. We concentrate on pointer assignment nodes, calculating the semantic effects (i.e., transfer functions) of the code at these nodes on the information reaching from an ICFG predecessor. Propagation is trivial through a node which is not a pointer assignment; such a node can neither create nor destroy aliases or pointer-types.

[10] We perform a lazy initialization of all predicates in constant time [LR92].

alias-implies-alias-thru-assign

may-hold (m, $\emptyset$, $<\!*q, a\!>$)
`n : p = q;`
may-hold (n, $\emptyset$, $<\!*p, a\!>$)

alias-implies-type-thru-assign

may-hold (m, $\emptyset$, $<\!*u, p\!>$)
`n : *u = new B;`
points-to-type (n, $\emptyset$, $<p \Rightarrow B>$)

type-implies-type-thru-assign

points-to-type (m, $\emptyset$, $<q \Rightarrow B>$)
`n : p = q;`
points-to-type (n, $\emptyset$, $<p \Rightarrow B>$)

type-implies-alias-thru-assign

points-to-type (m, $\emptyset$, $<q \Rightarrow B>$)
`n : p = q;`
may-hold (n, $\emptyset$, $<p \rightarrow mem_k, q \rightarrow mem_k>$)

Fig. 5. Example for intraprocedural propagation

3.3 Interprocedural Propagation

Propagation from `call` *node:* For non-virtual function calls, the corresponding `entry` node is easily determined. However if `call` represents a virtual call site, the *points-to-type* predicates involving the receiver at `call` determine the possible functions invoked. Each class associated with the receiver may correspond to a virtual function. For each `entry` so determined, we propagate information from `call` to `entry`. Having described the parameter binding functions already (see Figure 3), we can describe **alias-type-propagation-from-call** as setting *true* at `entry` those *may-hold* and *points-to-type* predicates corresponding to each element of the appropriate **bind, alias-bind** and **type-bind** sets. For example in Figure 3, $<r \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1> \in$ **alias-bind**($call_{c2}, entry_{C::bar}, <r \rightarrow d1 \Rightarrow B>$) results in **make-true**(*may-hold* ($entry_{C::bar}$, $<r \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1>$, $<r \rightarrow d1 \rightarrow b1, bar2 \rightarrow b1>$)).

Propagation from `exit` *node:* Suppose a pair APT holds at `exit` with assumption $AAPT$ at `entry`. Using the parameter binding functions we determine the `call`(s) responsible for imposing $AAPT$ at `entry`, and propagate APT only to the corresponding `return`(s). This pivotal role played by the binding functions allows us to propagate information along a good approximation of consistent paths.

alias-implies-alias-from-exit(`exit`,$AAPT$,*alias*): (i) If the entry assumption $AAPT$ is $\emptyset$, *alias* may hold at `exit` no matter which `call` invokes the function containing `exit`, as this alias pair is created solely due to the execution of the function. As a result, *may-hold* (`return`, $\emptyset$, *alias*) is made *true* for all `return`s corresponding to virtual or non-virtual `call`s invoking this function.

(ii) If `AAPT` is non-$\emptyset$, it implies that *alias* holds at `exit` if a `call` imposes $AAPT$ at `entry`. Suppose *may-hold* (`call`, $AAPT'$, AP) imposes $AAPT$ at `entry` through **bind**(`call`,`entry`,AP). Using this association, we **make-true**(*may-hold* (`return`, $AAPT'$, *alias*)). Also, for each *points-to-type* (`call`, $AAPT'$, PT) imposing $AAPT$ at `entry` through either **type-bind**(`call`,`entry`, PT) or **alias-bind**(`call`,`entry`,PT), we **make-true**(*may-hold* (`return`, $AAPT'$, *alias*)).

type-implies-type-from-exit(`exit`,$AAPT$,*pointer-type*): There are two cases similar to the previous function:

(i) If $AAPT = \emptyset$, **make-true**(*points-to-type* (`return`, $\emptyset$, *pointer-type*)),

(ii) For non-$\emptyset$ $AAPT$, **make-true**(*points-to-type* (`return`, $AAPT'$, *pointer-type*)).

4 Implementation Results

The results presented in this section represent our initial efforts to empirically demonstrate the contributions of the algorithm and assess its precision using a prototype implementation. The prototype is written in C and runs on a Sun SPARC-20 with 64MB main memory. We are using the MasterCraft C^{++} system of Tata Consultancy Services as the front end C^{++} parser for the implementation. Our aliasing and type determination algorithm reuses some code from the Landi-Ryder aliasing algorithm [LR92] with suitable modifications. Our prototype was engineered as a *proof of concept* implementation to test the effectiveness of our approach for virtual function resolution. The front end had many constraints to its functionality, which led to the use of test programs of 1000 lines of code or smaller. However, this is a limitation of the current implementation, not the algorithm. Thus, we are currently re-engineering our prototype for performance, with a new front end. We also expect to use a newer, optimized version of the aliasing algorithm, whose performance has been improved several orders of magnitude in space and time over that in the current prototype. This will allow us to analyze much larger programs (e.g., 10,000 lines of code).

Here, we present empirical results of analyzing 19 C^{++} programs obtained from various (publicly available) sources such as textbooks, demonstration programs accompanying a C^{++} compiler and undergraduate projects. We only analyzed programs using a library if a driver program was included.

Table 1 lists some characteristics of programs we analyzed, such as the lines of code (LOC), number of classes, number of functions, number of virtual functions and number of virtual calls. We also list the number of virtual call sites present in those functions which were found unreachable, number of aliases per ICFG node, number of pointer-types per ICFG node and analysis times. As stated above, our new implementation will improve the unacceptable times shown for *city*, *tree* and *employ*. In the remainder of this section (unless otherwise stated) we normalize the data with respect to virtual call sites in reachable functions. We also will list the programs by number (from Table 1) instead of by name.

Virtual function resolution Based on the distinct classes of objects pointed to by the receiver at a virtual call site, the algorithm determines which virtual functions in the inheritance hierarchy may be invoked from the call site. In Figure 6(a), we classify the reachable virtual call sites in terms of the number of virtual functions found invocable. Our results support the observation by Calder and Grunwald that although object-oriented libraries support polymorphism through virtual functions, the target of most indirect function calls can be

name	LOC	#classes	functions	virtual functions	virtual calls	%unreach virt calls	aliases per node	ptr-types per node	time (min:sec)
1. greed	968	2	47	9	17	35	2	2	0:31
2. garage	143	4	19	3	10	0	304	34	0:56
3. vcircle	142	3	16	4	5	0	0	1	0:01
4. office	213	5	12	4	4	0	49	10	0:04
5. family	109	3	22	3	3	0	4	3	0:02
6. FSM	98	4	15	2	1	0	6	2	0:01
7. deriv2	313	7	34	16	66	45	29	3	1:04
8. shapes	267	4	34	12	22	23	38	5	2:19
9. deriv1	192	7	31	13	28	29	20	4	0:15
10. objects	465	6	59	31	39	85	6	1	0:08
11. simul	339	7	54	12	7	15	3	1	0:03
12. primes	46	4	11	4	3	0	48	7	0:26
13. ocean	444	7	64	10	5	0	81	8	1:58
14. NP	31	2	7	2	6	0	0	9	0:01
15. city	519	11	67	2	1	0	322	12	13:27
16. tree	217	5	26	8	3	33	726	34	11:14
17. employ	894	10	58	25	4	0	213	14	7:42
18. life	178	5	21	8	2	0	49	3	0:26
19. chess	392	10	43	12	1	0	58	16	0:54

Table 1. Some characteristics of C^{++} programs analyzed

accurately predicted [CG94]. While their observation is based on execution profiles of programs, our results eliminate the dependence on profile data by using compile time analysis which accounts for *all* possible executions of the program. On examining the data, we ascertained that the virtual calls with non-unique resolution (i.e., calls with 2 or more possible methods) were indeed polymorphic at run-time, and that the non-uniqueness was never due to approximations in analysis. While such results are not guaranteed by our approach, they are encouraging.

We also wanted to find out if there was any relation between the program characteristics from Table 1 and virtual function resolution results in Figure 6(a). For example, it would have been interesting to find that the resolution results were better for programs with greater number of virtual calls or virtual functions. Figure 6(a) measures the utility of our algorithm solution for virtual function resolution - unique resolution is best and $>$ 3-way resolution is worst. The programs in Table 1 are sorted in decreasing order of solution utility. There are no corresponding trends in the program characteristics such as lines of code (LOC), number of virtual functions or the number of virtual calls. We conjecture that it is not these program characteristics but the programming style and the functionality of the program which influence the degree of polymorphism demonstrated by the virtual calls. However, a bigger data set containing larger programs is needed to draw definite conclusions about the program properties affecting virtual function resolution.

In Figure 6(b), we summarize how we obtained the unique virtual function resolutions in programs 1-13, shown as black bars in Figure 6(a). Looking only at those calls uniquely resolved in these 13 programs, we attribute the resolution of some of them either to the fact that there was only one choice of virtual function possible (discernible by examining the inheritance hierarchy) or to the use of the address of an object as receiver of the virtual call (thereby providing a unique type for the receiver). The percentage of virtual calls resolvable by the former method are shown as black bars; for the latter method the results are shown by diagonal-lined bars. The remainder of the virtual call sites were unresolvable with either of these two methods, but required our flow analysis technique. For these calls, we list the number of possible choices in the hierarchy (i.e., 2, ≥ 2) out of which we were able to select the unique function. Thus, Figure 6(b) shows those uniquely resolvable virtual calls for which our method proved stronger than optimized inheritance-based techniques.

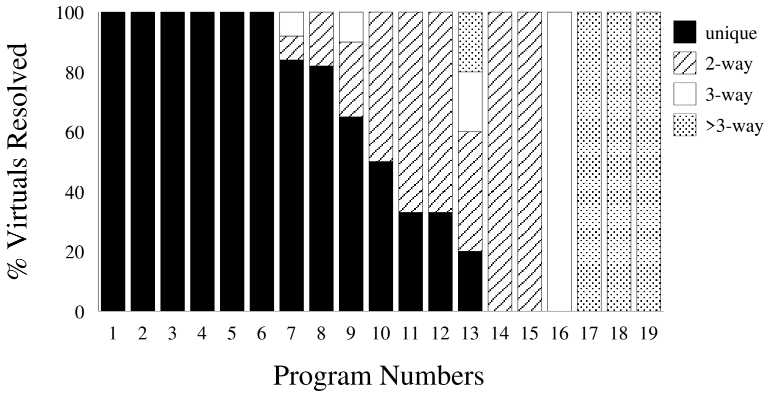
Invocable virtual functions at a call site and call graph construction Even when unique resolution is not possible, limiting the number of virtual functions invoked at a call site can help subsequent analyses, in that the side effects of the non-invocable functions can be safely eliminated in the context of that call. Given the potential disparity in various implementations of a virtual function, this can translate into improved precision and efficiency of analysis. Limiting the number of invocable virtual functions at a virtual call is relevant to tools which use branch prediction. In Figure 6(c), we report the average percentage of virtual functions found invocable out of those in the class hierarchy of the receiver type at a (reachable or unreachable) virtual call site. We create edges from a virtual call site only to the functions found invocable using resolution information, implicitly building a smaller and more precise call graph than the one built without such information. Therefore, the values in Figure 6(c) can also be interpreted as the size of virtual function portion of our call graph *vis à vis* that constructed using a naive approach. Viewing only the relevant aspects of the program (using a call graph browser which utilizes the analysis information) can be used for better program understanding. Eliminating non-invocable functions can also be used to prune the class libraries, since we can replace the implementation of a non-invoked virtual method by its signature, reducing the library source code as well as object code size.

As expected, the average percentage of invocable virtual functions was low (implying that a high percentage was eliminated) for those programs for which:

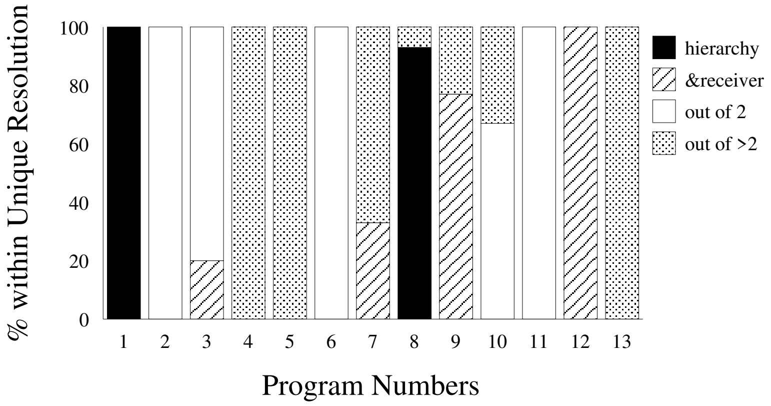
1. a high percentage of virtual calls were found unreachable (see Table 1), or
2. we obtained good virtual function resolution (see Figure 6(a)), or
3. most of the unique resolution was obtained from two or more candidate functions in the class hierarchy (see Figure 6(b)).

5 Conclusions

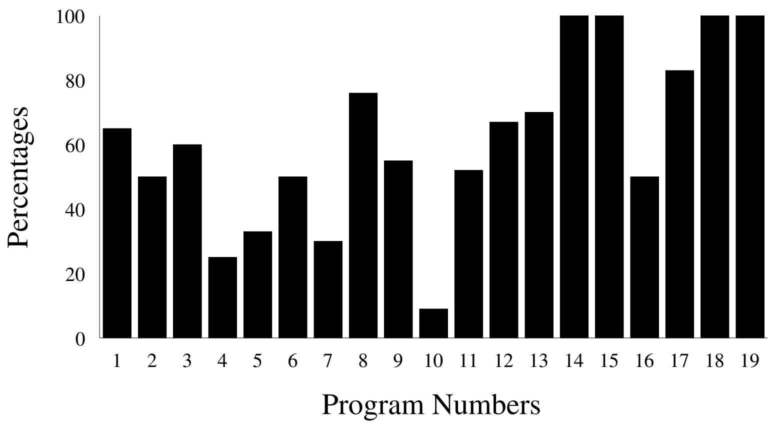
We have presented the first polynomial-time combined algorithm to perform program-point-specific, interprocedural type determination and aliasing for C^{++}. We have stated the theoretical difficulty of this problem and demonstrated the



(a) Percentage of virtual calls with 1,2,3,>3 invocable funcs



(b) Classification of unique resolution for programs 1-13 from (a)



(c) Average percentage of invocable virtual functions out of those in class hierarchy at virtual call

Fig. 6. Empirical Results

utility of its approximate solution in virtual function resolution on a modest dataset using a prototype implementation. Currently, we are redesigning our prototype for performance optimization and to be able to analyze larger C^{++} programs. We also are considering work on useful analyses for other compiler transformations, debugging and testing in a C^{++} programming environment.

References

[CG94] B. Calder and D. Grunwald. Reducing indirect function call overhead in C^{++} programs. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[CGZ95] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioural differences between C and C^{++} programs. *Journal of Programming Languages*, 2:313–351, 1995.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

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

[Pan96] H. Pande. *Compile Time Analysis of C and C^{++} Systems*. PhD thesis, Rutgers University, May 1996.

[PR94a] H. Pande and B. G. Ryder. Static type determination for C^{++}. In *Proceedings of the Sixth USENIX C^{++} Technical Conference*, pages 85–97, April 1994.

[PR94b] H. D. Pande and B. G. Ryder. Static type determination and aliasing for C^{++}. Laboratory for Computer Science Research Technical Report LCSR-TR-236, Rutgers University, December 1994.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

This article was processed using the LaTeX macro package with LLNCS style