On Interprocedural Data Flow Analysis for Object Oriented Languages

Mario Südholt* and Christoph Steigner**

Abstract. *As object oriented languages ease software construction significantly, these languages are very promising candidates for parallelizing compilers. To combine the advantages of object oriented programming with the power of parallel processing two major problems have to be solved:* the virtual function *and* the class scope problem. *We present solutions to these problems and exemplify them by extending a fast interprocedural data flow analysis algorithm.*

Keywords: Object oriented language, interprocedural data flow analysis, virtual function, class scope

1 Introduction

Object oriented languages have two important advantages over conventional (imperative sequential and parallel) programming languages:

- *Code reuse* by using existing class hierarchies greatly increases efficiency of the programming process.
- The concept of an object encapsulating its internal state represents a particularly *natural model for distribution and parallel execution* on distributed memory architectures.

Hence, object oriented languages gain more and more influence in the field of programming languages and should therefore be available for programming of multiprocessor environments.

Interprocedural data flow analysis is a key issue in the field of compiler optimization and parallelizing compiler technology as described elsewhere (see [3], [4]). Object oriented languages place a still higher demand on the interprocedural data flow analysis phase of a (parallelizing) compiler, because encapsulation of the internal state of an object trades off with a higher number of procedures and object methods used. It is therefore of imminent importance to be able to use fast interprocedural data flow analysis algorithms to analyze object oriented languages.

The fastest known data flow analysis algorithms for the interprocedural alias-free flow-insensitive side-effect problem have a time complexity which is linear in the size of the call graph. As we will show in section 2 the *virtual function problem* of object oriented languages needs a more sophisticated approach in order to meet this time bound in the presence of virtual functions and late binding.

* Technical University of Berlin, Institute of Applied Computer Science, Franklinstraße 28/29, D - 1000 Berlin 10, Secretariat: FR 5/13

** University of Koblenz, Institute of Computer Science, Rheinau 3–4, D - 5400 Koblenz

A second problem characteristic to object oriented languages is the *class scope problem* which arises as a consequence of the introduction of new scoping rules. As illustrated in section 3 common techniques to cope with problems based on scoping rules (e. g. nested procedure scope) are not applicable to the class scope problem in object oriented languages.

The problems and solutions developed in the next two sections are illustrated in section 4 by extending Cooper's and Kennedy's algorithm (see [1]) for interprocedural data flow analysis to object oriented languages.

The terminology of the object oriented language C++ is used in this paper and the examples are written in C++. This means, in particular, that *virtual functions* are functions which are bound at run time by late binding.

2 The virtual function problem

Almost all data flow analysis algorithms are based on the use of different forms of flow graphs. The complexity of algorithms working on these graphs is usually stated in terms of traversals of edges and vertices. The fastest known algorithms operate in time linear to the sum $N + E$, where N stands for the set of vertices and E the set of edges.

In the field of interprocedural data flow analysis the graph sizes depend mainly on two variables, the number of procedure calls (in case of the call graph) and formal parameters (e. g. in Cooper's and Kennedy's binding-multigraph which is defined in section 4).

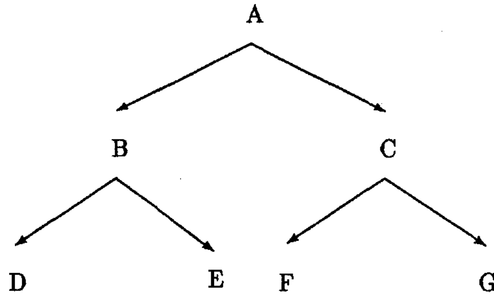
Using object oriented languages these two variables may grow exponentially with the depth of the inheritance graph, which is illustrated by the example shown in Fig. 1. Figure 1.a) shows a simple class hierarchy and Fig. 1.b) shows the corresponding implementation of various classes and functions. Only the characteristics essential to the virtual function problem are shown in the implementation. It should be noted that all seven classes in the class hierarchy implement the virtual function `f(a, b)`. Figure 1.c) shows an example application in which the function `caller` calls the virtual method `var->f(a, b)`.

If the compiler cannot restrict the class type of object `*var`, it has to assume that the pointer `var` can point to an object of class `A` and all of its subclasses (`B`-`G`). In this case the virtual function call `var->f(a, b)` adds seven edges to the call graph and not only one edge as in non-object oriented languages. Even a multitude more will be introduced in the binding-multigraph. It is evident, that the resulting *complexity of the overall graph traversal is exponential in the depth of the inheritance graph in the worst case.*

Since new classes derived from this class hierarchy augment the depth of the inheritance graph, *code reuse* (which is performed just by deriving new classes) does give rise to this problem.

This problem can be solved by using the general algorithm presented in figure 2.

There certain base classes serve as *representants* for the derived partial graph (of the inheritance graph) which includes the representants and all classes derived from them.



(a) Example class hierarchy (→ : "is base class of")

```
class A { ... virtual void f(int * a, int * b); ... };

class B : A { ... virtual void f(int * a, int * b); ... };
class C : A { ... virtual void f(int * a, int * b); ... };

class D : B { ... virtual void f(int * a, int * b); ... };
class E : B { ... virtual void f(int * a, int * b); ... };
class F : C { ... virtual void f(int * a, int * b); ... };
class G : C { ... virtual void f(int * a, int * b); ... };
```

(b) Example implementation of class hierarchy (a)

```
void caller(int * a, int * b) {
    A * var;
    ...
    var->f(a, b);  // call to virtual f()
    ...
}
```

(c) Example function using class hierarchy (a)

Fig. 1. Virtual functions may cause exponential behaviour in the size of the inheritance graph

To achieve this, the class hierarchy's data flow analysis graph will be investigated by means of conventional flow analysis algorithms. All relevant information about virtual functions will then be merged with the representants' information.

At compile time of the overall application (function `caller` in example 1) each call of a virtual function does hence add only edges from the calling procedure to the representants' virtual procedure. This procedure is most precise, if the represented virtual functions behave similar, but this is the case normally, because the virtual function mechanism has been designed especially for this purpose.

1. At compile time of a given class hierarchy:
   (a) Determine all base classes which are used as ***representants.***
   (b) Compute the data flow information concerning all procedures and methods of the class hierarchy.
   (c) Propagate all information regarding virtual functions of derived classes to the representants' vertex and merge it with the information already computed there.
2. At compile time of the overall application use the information of the representants' vertex where any information about virtual functions of the representant or any class derived from them is needed.

Fig. 2. General algorithm to solve the virtual function problem

Strong time bounds can be met by exploiting the fact that the use of representants instead of references to all virtual functions affected in step 2 of algorithm 2 can already be used incrementally in step 1.b of the algorithm. By using this technique we avoid that the virtual function problem applies during step 1.b.

An example how this technique can be used algorithmically is shown in section 4, where the general algorithm of figure 2 is specialized.

3 The class scope problem

A second problem is a consequence of the fact that most object oriented functions introduce a new sort of scope, the *class scope.* Variables which are declared to be private to an object (i. e. they can only be accessed directly by the object's methods) belong to a class scope. Modifications to these variables can be lifted to modifications of their respective objects in view of efficiency in certain cases.

This is a problem, since most efficient data flow analysis algorithms exploit the specific features of scoping rules. Cooper's and Kennedy's algorithm, for example, is applicable in its purest form only to languages with *global-local scoping* (as in C, FORTRAN). They indicate, however, a way to adapt their algorithm to languages with *nested-procedure scope* (such as Pascal). This is done by regarding the nested procedures' bodies as an extension of the enclosing procedure. This approach cannot be used with class scoping, because objects cannot be necessarily assigned to specific procedures.

In particular, there is the following problem with ordinary data flow analysis algorithms based on a call graph or other data flow graphs: *Usage or modification of an object and its private variables has to be distinguished.* This is necessary, because at a certain level within the call graph knowledge about the internal variables of an object gives no more useful information and can hence be forgone in view of efficiency. This is a direct consequence of objects being the means for providing encapsulation.

These problems can be solved by annotating each vertex of the (call) graph with both the object and the internal variables changed and providing a function which "lifts" modifications of internal variables to modifications of their respective

objects after which the internal details can be omitted. After lifting the information concerning object variables is ommitted. Since the number of object variables largely exceeds the number of object this techniques speeds up data flow analysis.

This solution is also incorporated in the algorithm presented in the next section.

4 An interprocedural data flow analysis algorithm

Cooper's and Kennedy's algorithm (published in [1]) solves the alias-free flow-insensitive side-effect analysis problem by splitting it into two subproblems, which are solved separately. First side-effects due to formal reference parameters (formal reference problem) and then the problem of side-effects due to global variables (modified global variable problem) is solved.

To solve the *formal reference problem* they introduce the *binding-multigraph*, $\beta = (N_\beta,\ E_\beta)$. N_β is the set $\{fp_p^i\}$ of all formal parameters at all call sites. There is an edge $(fp_p^i, fp_q^j) \in E_\beta$ if there is a *binding event*, that is p calls q and the ith formal parameter of p gets bound to the jth formal parameter of q at the call site. This definition is illustrated by Fig. 3 which shows the binding multigraph for the example in Fig. 1.

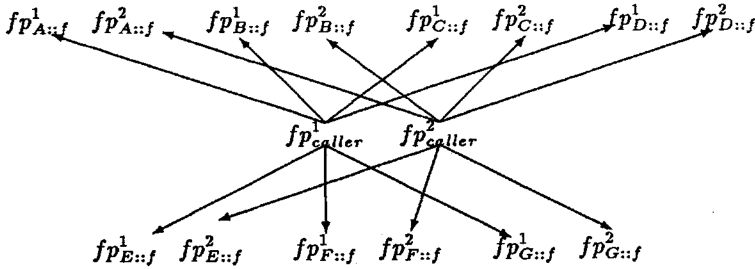


Fig. 3. Example binding-multigraph

They show that the size of the binding-multigraph is bound by a constant to the size of the call graph. The *virtual function problem*, however, applies here, because formal parameters of virtual functions may cause an exponential growth of the binding-multigraph.

The solution stated in section 2 is perfectly applicable to this problem, since the formal parameter problem constitutes a problem which is invariant to the surrounding context in which functions are called. This means, that the representants which are calculated have to be updated only if the class hierarchy is changed, but not if the same class hierarchy is used in another context. In example 3 twelve edges are removed if the representant chosen is the base class A, because the only remaining edges are $fp^1_{caller} \rightarrow fp^1_{A::f}$ and $fp^2_{caller} \rightarrow fp^2_{A::f}$. By using this technique we can achieve the linear time bound with the algorithm shown in figure 4.

1. Designate the base classes which are used as representants. In the simplest case all base classes which have virtual methods can be used.
2. Compute all modifications to formal reference parameters within the class hierarchy by using Cooper's and Kennedy's algorithm.
3. Use Cooper's and Kennedy's algorithm to solve the formal reference problem for the whole application.

Fig. 4. Solving the formal reference parameter problem

The most important observation with this algorithm is that steps 1. and 2. have to be done only once for a class hierarchy (e. g. system libraries etc.). The whole algorithm certainly meets the linear time bound on the call graph, for the following reasons:

- Depending on the criteria to select the representants the first step can be done in constant time.
- Step 2 applies Cooper's and Kennedy's algorithm to the methods in the class hierarchy — that is a small subproblem compared to the overall application. In addition, representants can already be used while analyzing the class hierarchy as stated in section 2.
- Step 3 only requires linear time as shown in [1], because the virtual function problem does not apply while using representants.

After determination of side effects due to the formal reference parameter problem, side effects due to global variables are determined. Here the virtual function problem is also avoided by a specialized version of algorithm 2. In the following a presentation is given how their algorithm to solve the *global modifications problem* can be adapted to object oriented languages. Cooper and Kennedy observed that sets of modified variables can efficiently be computed by regarding the strongly connected components[3] (henceforth SCC) of the call graph. An adaption of Tarjan's algorithm (described in [2]) can thus be used for computation of the modified global variable problem. Once all members of a SCC are found they propagate all side effects to global variables between these members.

The class scope problem can then be solved using the following functions and variables:

OV[p]: These sets are bit vectors used to register modifications to object variables for each procedure p.

$IMOD_o^+[p]$: These sets hold all object variables which are modified either directly in procedure p or due to side effects to reference parameters within p and the functions which are ever called in p. These sets have been determined by solving the formal reference problem as described in algorithm 4.

ClassScope(): This function yields the subset of all modifications of object variables in *OV[p]* which have to be lifted to modifications of objects in *GMOD[p]*. To

[3] Two vertices are members of a strongly connected component iff there are two destinct paths linking them.

determine the object variables which have to be lifted, it uses the root of the actual strongly connected component and the object of the procedure considered.

The adaption of the algorithm to object oriented languages with scoping rules as described above is now rather straightforward. Modifications to object variables are recorded in sets *OV[p]*, which are initialized to $IMOD_o^+[p]$. They are updated along with modifications to other variables while treating edges which link two different SCCs in the call graph. If all vertices of a SCC are found, the algorithm lifts modifications to object variables to modifications of their respective objects using the function *ClassScope()*.

Cooper's and Kennedy's correctness proof has been extended to our approach by extending their recursive flow equation scheme and adapting their proof accordingly. The proof ensures that all variables modified as side effects of a strongly connected component (henceforth *s*; *s* be closed[4] in line (26) of the algorithm) are correctly propagated to all other strongly connected components which can reach *s*.

5 Conclusion and future work

We have shown that conventional data flow analysis algorithms are not applicable for object oriented languages because of the *virtual function* and *class scope problem*. To solve these problems new techniques have been presented which have been proven useful in the framework of conventional data flow analysis algorithms.

Further work will be done in order to apply the techniques presented here to a greater number of analysis techniques (especially in the field of analyzing array accesses). The overall approach will be enhanced in order to use our techniques in the more general framework of a parallelizing compiler for object-oriented languages. Efficiency and granularity constraints in relation to object sizes and communication will be investigated with regard to different target multiprocessors.

References

1. Keith. D. Cooper, Ken Kennedy: *"Interprocedural Side-Effect Analysis in Linear Time"*; Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 22 – 24, 1988
2. Jürgen Ebert: *"Effiziente Graphenalgorithmen"*; Studientexte, Akademische Verlagsgesellschaft, 1981
3. Laurie J. Hendren, Alecandru Nicolau: *"Parallelizing Programs with Recursive Data Structures"*, in: IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990
4. Paul Havlak, Ken Kennedy: *"An Implementation of Interprocedural Bounded Regular Section Analysis"*, in: IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 3, July 1991

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

[4] A strongly connected component is *closed* if its root has been identified and all its members have been popped off the stack.