The Impact of Interprocedural Class Analysis on Optimization

David Grove
Department of Computer Science and Engineering
University of Washington

Abstract
The runtime performance of object-oriented languages often suffers due to the overhead of dynamic dispatching. In order to make these languages competitive with traditional languages, optimizing compilers attempt to eliminate as many of the dynamic dispatches as possible. A variety of local and intraprocedural techniques have been developed to do this, but they can be ineffective when they are unable to statically bind and inline a message send. To enable better analysis across non-inlined message sends, interprocedural analysis is required. In this paper we describe a simple algorithm for interprocedural class analysis and empirically evaluate its effectiveness as a tool for program optimization. We demonstrate that interprocedural class analysis can substantially decrease application execution times when compared to applications optimized with only intraprocedural techniques. *

1 Introduction
Applications written in object-oriented languages often perform poorly when compared to equivalent applications written in traditional languages. One of the major sources of this poor runtime performance is the heavy usage of dynamic dispatches, or message sends, which are more expensive than traditional procedure calls. This problem is particularly pronounced in pure object-oriented languages, which conceptually implement even the most basic operations, such as arithmetic operations and control structures, as message sends. In addition to the problems of dynamic dispatching, good object-oriented design encourages code factoring, which tends to distribute code that would have been a single procedure in traditional languages across multiple procedures. This code scattering tends to decrease the length of code sequences, which often lessens the effectiveness of traditional compiler optimizations. In order to make the performance of object-oriented languages competitive with traditional languages, optimizing compilers strive to eliminate the vast majority of these dynamic dispatches through static binding and subsequent inline-substitution. This eliminates both the direct cost of message send overhead and the indirect costs due to the lessened effectiveness of traditional compiler optimizations on shorter code sequences.

However, in order to statically bind a message send, the compiler must be able to determine that only a single method can be invoked at runtime. This requires the compiler to determine the classes of the arguments to the message, since this information affects the result of method lookup. To aid in this process, several local and intraprocedural techniques, such as class prediction [Deutsch & Schiffman 84, Chambers et al. 89, Hölzle 94] and iterative class analysis [Chambers & Ungar 90], have been developed. The goal of these techniques is to find, for each expression in the program, the smallest possible set of classes such that any runtime value of the expression will be an instance of one of the classes in its associated set. This mapping of expressions to sets of classes can then be used to perform compile time method lookup to determine the set of methods that could be invoked as a result of some message send. If this set of possible methods has exactly one element, then the message send can be statically bound. These techniques can be very effective, but are often hampered by the

* The IBM contact for this paper is Karen Bennet, Centre for Advanced Studies, IBM Canada Ltd., Dept. 21/894 844 Don Mills Road, North York, Ontario M3C 1V7.
conservative assumptions they are forced to make when either a message send cannot be statically bound or a statically bound call site is undesirable to inline. In particular, intraprocedural analyses are unable to determine the set of classes that should be bound to the return value of a non-inlined message send. This lack of knowledge can then hinder downstream optimizations, since the result of one send expression is often used as an argument to another. To address this limitation of the intraprocedural techniques, it seems natural to investigate interprocedural approaches. For example, if the compiler knew, for each method in the program, the set of classes that could be returned as a result of invoking the method, then a failure to statically bind a message send would not be a serious problem. Several algorithms have been proposed for determining this information [Palsberg & Schwartzbach 91, Oxhøj et al. 92, Agesen et al. 93, Plevyak & Chien 94, Agesen 95].

In this paper, we describe a simple algorithm for interprocedural class analysis, similar in spirit to Palsberg and Schwartzbach’s basic algorithm, and empirically assess its effectiveness as a tool for program optimization. The main contributions of this paper are:

• We demonstrate that even though the results of the basic algorithm have been shown to be substantially less precise than the results of the latter, more sophisticated, algorithms, they are precise enough to enable substantial performance improvements.
• We have successfully applied interprocedural class analysis to applications that are an order of magnitude larger than those used to evaluate previous work; Plevyak and Chien’s largest benchmark is 2000 lines, Agesen’s is 11,000, and ours is 52,000. Our results indicate that scalability of the analysis is a serious concern; successfully analyzing a small program of a few thousand lines is a much easier problem than attempting to efficiently analyze a much larger application.

In the next section we briefly review existing intraprocedural techniques to eliminate dynamic dispatches. In Section 3 we present a simple algorithm for interprocedural class analysis, and then evaluate its performance in Section 4. Finally, we discuss related work and offer some conclusions.

2 Intraprocedural Techniques

Receiver class prediction and iterative class analysis are both effective intraprocedural techniques for converting dynamic dispatches into static calls. As illustrated in Figure 1, receiver class prediction is a fairly simple local code transformation. In this example, the compiler has transformed a send of the area message into a test to see if x is a Rectangle followed by either a static call to Rectangle::area or, if x is not a Rectangle, a send of the area message. The decisions of which messages to predict and what classes to predict them for can be made by utilizing either a small, hard-wired set of message/class pairs [Deutsch & Schiffman 84, Chambers & Ungar 90], or by utilizing profile information [Hölzle 94, Grove et al. 95]. Since receiver class prediction is at best an educated guess by the compiler, a full dynamic dispatch must be preserved along the false branch of the test.

The compiler could replace a dynamic dispatch with a static call without the overhead of a class test if it could determine that for all the possible receiver

Before Class Prediction

    // send of the area message
    y := x->area();

After Class Prediction and Inlining

    if(x is Rectangle) {
        y := x.length * x.width;
    } else {
        y := x->area() // send area
    }

Figure 1: Receiver Class Prediction
classes, exactly one method will be invoked as a result of the dynamic dispatch. One means of determining this is iterative class analysis, which uses a standard iterative dataflow approach to determine the set of classes that correspond to program expressions [Chambers & Ungar 90]. The analysis maintains a mapping of variables to sets of classes, similar in spirit to the mapping of variables to constants that is used to perform constant propagation. At the start of the analysis, the mapping is initialized by binding the formal parameters of the method’s receivers to the appropriate sets of classes. Certain program structures, such as object creations, literal expressions, and class tests inserted by class prediction, allow the analysis to add useful information to the mapping. An assignment of x to y causes y to be mapped to the same set of classes as x. When a dynamically dispatched message send is encountered, the compiler utilizes its mapping of variables to sets of classes to perform compile time method lookup, thus computing the set of possibly invoked methods. If only a single method could be invoked, then the dynamic dispatch can be replaced with a static call to the appropriate method, which may subsequently be inlined. However, if a message send is not inlined, then a purely intraprocedural analysis will not be able to determine the set of classes that might be returned as the result of the non-inlined call. This inability may have negative downstream effects, since it is a common programming style in object-oriented languages to “chain” together a sequence of message sends, using the result of one invocation as an argument to the next.

3 Interprocedural Class Analysis

Interprocedural class analysis extends intraprocedural class analysis by analyzing the entire program, thus allowing a more precise treatment of non-inlined calls. Instead of simply flowing sets of classes through a single method’s control flow graph, information is propagated through the entire program’s call graph.

Many standard interprocedural analyses work by traversing a program call graph and computing some set of information for each node (method) in the call graph. Unfortunately, this straightforward approach is not possible for this particular interprocedural problem, because a useful conservative call graph cannot be constructed in advance. In order to construct a useful call graph, it must be possible to determine with reasonable precision which methods might be invoked at a dynamically dispatched call site; this in turn requires accurate information about the mapping of program variables to sets of classes, which requires interprocedural class analysis, which needs an accurate call graph. This circularity can be broken by simultaneously performing interprocedural class analysis and constructing the program call graph. We use the intermediate results of interprocedural class analysis to build the call graph, and use the partially built call graph to perform interprocedural class analysis, gradually refining both pieces of information until the analysis is complete. This approach is possible because the sets of classes are monotonically increasing.

Before describing the algorithm in more detail, we define the basic data structures used to implement it.

- **Monitored sets**: A monitored set consists of a set of classes, and a list of call graph nodes that depend on the information represented by the set. Whenever a node uses the information in the set, it is added to the set of nodes that depend on the set. Whenever a new element is added to the set, all of the nodes that depend on the set are added to the work-queue for reanalysis.

- **Call graph nodes**: Each lexical scope in the program (each method, closure, and the global scope) corresponds to a single node in the call graph. Each node contains the following information:
  - A table mapping the node’s formal parameters and local variables to monitored sets of classes. Each variable is represented by a single monitored set, which is initially empty.
  - A monitored set of classes, initially empty, that represents the node’s return value.
  - If the node’s scope is nested, then a pointer to the call graph node that represents the lexically enclosing scope. This pointer can
be used to walk the lexical chain to resolve variable references.

• A pointer to the abstract syntax tree (AST) corresponding to the scope represented by the node.

Global variables are represented as local variables of the global scope’s call graph node. Each instance variable is modeled as a pair of accessor methods (get and set) that share a single monitored set that represents the contents of the instance variable.

After an initialization phase, the algorithm begins by adding the top-level node to the work-queue. Nodes are then removed from the queue and processed until the queue is empty, at which point the analysis is complete. Processing a node entails traversing the node’s AST, performing a slightly modified version of intra-procedural class analysis. Those parts of this analysis that differ from standard intra-procedural class analysis are detailed below.

• **Variable reference**: To determine the set of classes corresponding to a variable reference, the analysis starts at the call graph node in which the reference occurred, and searches up the lexical chain of call graph nodes until it finds a definition of the variable. If a definition is found, then the monitored set of classes corresponding to that definition is used as the value of the reference, and the node in which the reference occurred is added to the set of dependents of the monitored set of classes. If no definition is found, then an empty set is used as the value of the reference (this corresponds to a runtime error of referencing an undefined variable).

• **Assignment statement**: The lexical chain of call graph nodes is searched to find a definition (and corresponding monitored set) for the left-hand side of the assignment. If the set of classes computed for the right-hand side of the assignment contains any classes not already present in the left-hand side’s monitored set, then they are added to it, and all dependent call graph nodes are enqueued for reanalysis.

• **Return statement**: The set of classes corresponding to the return statement’s value is added to the monitored set of classes that represents the current call graph node’s return value. This may result in call graph nodes being enqueued for reanalysis, if the size of the set increases.

• **Message send**: The sets of classes that correspond to the arguments to the send are used to perform compile time method lookup. The result of this lookup is a collection of call graph nodes; each call graph node represents a method that could be invoked as a result of the message send. For each possibly invoked call graph node \( n \), we compare the argument sets of classes to the sets of classes corresponding to \( n \)'s formal parameters. If the argument sets are not all subsets of the formals sets, then the missing classes are added to the formals set, and \( n \) is enqueued for reanalysis. We then add a dependency link between \( n \)'s return value set and the current call graph node, and add all the classes in \( n \)'s return value set to the set of classes that is being computed as the result of the message send.

This algorithm produces several kinds of information that can be used for optimization. It computes sets of classes for all program variables, for the return values of methods, and for the contents of all instance variables. The information about return values can be used to determine the results of non-inlined message sends. Information about variables can be used to make less pessimistic assumptions about what can happen to variables defined in lexically enclosing scopes across procedure calls. The algorithm also produces a call graph that can be used as the input to other inter-procedural analyses.

As presented, the algorithm assumes that the complete source of the program is available for analysis. This may not be true if the program uses precompiled libraries. The algorithm can be modified to handle this relatively easily, as long as library routines do not make arbitrary calls back into the application program. If a library routine is

---

\*We actually use a filtered set of classes that contains only those classes that could actually result in invoking the method being considered. For example, if the argument set of classes was \( \{\text{Square}, \text{Circle}, \text{Rectangle}\} \) and the method being considered was defined on \( \text{Quadrilaterals} \), then the set of applicable classes would be \( \{\text{Square}, \text{Rectangle}\} \).
We first ran the analysis on each of these benchmark programs and measured the cardinality of the sets of classes that were computed by the analysis. This is a metric that has been commonly used in previous work to compare algorithms for interprocedural class analysis. In general, the more classes there are in the set, the less likely it is that the information can be used to optimize the program. Sets containing a single class can always be used for optimization; sets with two or three elements are often useful; and sets with more elements are frequently not useful.

Table 2 shows the cardinality distribution for all the sets of classes encountered during the analysis, in the absence of other information the algorithm simply needs to assume that all closures/function pointers passed as parameters to the routine are called with arbitrary arguments, and that the return value of the library routine is the set containing all classes.

### Table 2: Cardinality Distribution of All Sets of Classes

<table>
<thead>
<tr>
<th>Program</th>
<th>1 Class</th>
<th>2 Classes</th>
<th>3 Classes</th>
<th>4 Classes</th>
<th>5 Classes</th>
<th>6 or more</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richards</td>
<td>51%</td>
<td>15%</td>
<td>1%</td>
<td>1%</td>
<td>2%</td>
<td>26%</td>
</tr>
<tr>
<td>Deltablu</td>
<td>60%</td>
<td>13%</td>
<td>1%</td>
<td>1%</td>
<td>2%</td>
<td>24%</td>
</tr>
<tr>
<td>InstrSched</td>
<td>59%</td>
<td>10%</td>
<td>2%</td>
<td>1%</td>
<td>1%</td>
<td>26%</td>
</tr>
<tr>
<td>Typechecker</td>
<td>59%</td>
<td>11%</td>
<td>2%</td>
<td>3%</td>
<td>1%</td>
<td>24%</td>
</tr>
<tr>
<td>Compiler</td>
<td>56%</td>
<td>9%</td>
<td>1%</td>
<td>3%</td>
<td>1%</td>
<td>29%</td>
</tr>
</tbody>
</table>

We expected that the algorithm would be less effective when applied to larger programs, since they make heavier usage of polymorphic data structures, which are a known weak point of the basic algorithm. However, the cardinality distributions were surprisingly similar for all five benchmarks. This seems to suggest that the effectiveness of the algorithm is more or less independent of program size. However, if we examine the execution speeds shown in Figure 2, we can see that the analysis was in fact more effective on small programs than on large programs. Thus, the cardinality metric is not a good predictor of application runtime improvements.

### 4 Experimental Evaluation

The algorithm described in the previous section has been implemented as part of the Vortex optimizing compiler for object-oriented languages. To evaluate its effectiveness, we used the suite of Cecil programs described in Table 1. Cecil is a pure object-oriented language [Chambers 93].

### Table 1: Cecil Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines(^a)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richards</td>
<td>400</td>
<td>Operating systems simulation</td>
</tr>
<tr>
<td>Deltablu</td>
<td>650</td>
<td>Incremental constraint solver</td>
</tr>
<tr>
<td>InstrSched</td>
<td>2,400</td>
<td>MIPS global instruction scheduler</td>
</tr>
<tr>
<td>Typechecker</td>
<td>17,000(^b)</td>
<td>Cecil static typechecker</td>
</tr>
<tr>
<td>Compiler</td>
<td>45,000(^b)</td>
<td>Vortex optimizing compiler</td>
</tr>
</tbody>
</table>

\(^a\) Not including 8,500 line standard library
\(^b\) The typechecker and compiler share approximately 12,000 lines of code

We first ran the analysis on each of these benchmark programs and measured the cardinality of the sets of classes that were computed by the analysis. This is a metric that has been commonly used in previous work to compare algorithms for interprocedural class analysis. In general, the more classes there are in the set, the less likely it is that the information can be used to optimize the program. Sets containing a single class can always be used for optimization; sets with two or three elements are often useful; and sets with more elements are frequently not useful. Table 2 shows the cardinality distribution for all the sets of classes.

We expected that the algorithm would be less effective when applied to larger programs, since they make heavier usage of polymorphic data structures, which are a known weak point of the basic algorithm. However, the cardinality distributions were surprisingly similar for all five benchmarks. This seems to suggest that the effectiveness of the algorithm is more or less independent of program size. However, if we examine the execution speeds shown in Figure 2, we can see that the analysis was in fact more effective on small programs than on large programs. Thus, the cardinality metric is not a good predictor of application runtime improvements.
To assess the impact of interprocedural class analysis on application performance, we compiled the benchmark programs using the following compiler configurations:

- **unopt**: No optimizations were performed.
- **intra**: Intraprocedural class analysis and inlining, hard-wired class prediction, extended message splitting [Chambers & Ungar 90], class hierarchy analysis [Dean et al. 95], closure optimizations, common subexpression elimination, constant folding and propagation, and dead assignment elimination.
- **intra + inter**: **intra** augmented with interprocedural class analysis.
- **intra + profile**: **intra** augmented with profile-guided receiver class prediction [Hölzle 94, Grove et al. 95].
- **intra + profile + inter**: **intra + profile** augmented with interprocedural class analysis.

Figure 2 shows the execution speeds of the benchmark programs, compiled with these configurations. For all five benchmarks, there was a substantial speedup (25% to 144%) when the standard intraprocedural optimizations were augmented with interprocedural class analysis. However, the improvements were much more modest, especially on the larger benchmarks (1% to 7%), in configurations that included profile-guided receiver class prediction. The greater improvement seen in the smaller programs is explained by their limited usage of polymorphic functions and data structures. The basic interprocedural class analysis described in the previous section maintains only a single abstract representation of each function and instance variable during the analysis. When many clients utilize a common abstraction for different purposes, such as a generic linked list or hash table class, serious imprecisions may occur, since all usages are smeared together during the analysis. In the small programs, this is not a serious problem, since most abstractions have only one or two clients. The solution to this precision problem is to modify the analysis to maintain multiple abstract representations for a single source method or instance variable, thus allowing them to be analyzed separately for each different use. This separation into multiple abstract representations is known as splitting. Much of the related work discussed in the following section has focused on different schemes for deciding what methods to split and how many abstract representations they should be split into, since it is too expensive (in terms of analysis time) to split everything.

Unfortunately, analysis time does not scale gracefully with program size. For the three small programs, analysis times ranged from 30 to 210
7 seconds, 50 minutes were required to analyze the typechecker, and analyzing the compiler consumed 8.5 hours (all times are CPU times on an otherwise lightly loaded SPARC 20/61). One of the primary causes of the rapid and non-linear growth in analysis time is that the imprecision of intermediate analysis results forces the algorithm to analyze many non-realizable paths in the call graph. By utilizing splitting to selectively increase the precision of the analysis, we expect that not only will the analysis results be more precise, thus enabling more effective optimization, but that analysis times will decrease substantially, since many fewer non-realizable paths will be analyzed. Current work includes developing new algorithms to guide splitting decisions, and implementing several of the algorithms described in the next section for comparison purposes.

5 Related Work

A problem closely related to interprocedural class analysis is the construction of a call graph in a language that allows functions to be passed as parameters. Just as in interprocedural class analysis, the targets of some call sites cannot be accurately determined without tracing the flow of certain abstract values through the program. Early work in this area was done by Shivers who developed the k-CFA family of algorithms, where k is a parameter that controls the degree of splitting [Shivers 91]. The 0-CFA algorithm smears all call sites together and performs no splitting; it has similar precision characteristics as the algorithm described in section 3. A k-CFA algorithm splits all call chains above the leaves to a height of k. In these algorithms, all call sites in the program are split to the same depth; one cannot choose to split some call sites and not split others. Hall and Kennedy developed an efficient work-queue-based algorithm [Hall & Kennedy 92] that achieves the same precision as Shivers’ 0-CFA algorithm. They observe that since their algorithm is mainly intended for the analysis of Fortran programs, a more precise algorithm is not required in practice. Steensgaard has developed an adaptive algorithm that achieves extremely precise results without the large amounts of useless work that an equally precise, non-adaptive algorithm would perform [Steensgaard 94]. The algorithm automatically determines the minimal level of splitting necessary to eliminate all imprecisions except those introduced by cycles in the call graph.

Palsberg and Schwartzbach developed two variants of an algorithm for performing interprocedural class analysis for a Smalltalk-like language [Palsberg & Schwartzbach 91, Oxhøj et al. 92]. Their algorithms work by first duplicating all methods to remove inheritance and then creating and solving a system of constraints. Their basic algorithm has the same precision as Shivers’ 0-CFA, and they describe an extension that corresponds to 1-CFA. Aagesen has developed an algorithm derived from the Palsberg and Schwartzbach algorithm that uses a set of heuristics to adaptively decide which call sites should be split and to what level they should be split [Aagesen et al. 93]. More recently, he has developed an algorithm that precisely and efficiently analyzes polymorphic functions, but does not address the problem of polymorphic data structures [Aagesen 95]. Plevyak and Chien have also developed an algorithm that adaptively determines where splitting is needed to remove imprecisions in the analysis results [Plevyak & Chien 94]. Their algorithm works by iteratively performing the analysis, determining where splitting would improve the analysis results, and reapplying the analysis until either no imprecisions remain or a fixed number of iterations have been performed. Pande and Ryder describe an algorithm for performing interprocedural class analysis for a subset of C++ that approaches the problem from the perspective of interprocedural alias analysis [Pande & Ryder 94]. Aagesen and Hölzlze compare the effectiveness of interprocedural class analysis and profile-guided receiver class prediction for the optimization of Self programs, but do not report on the impact of combining the two techniques [Aagesen & Hölzlze 95].

6 Conclusions

We have presented a simple algorithm for interprocedural class analysis and have empirically evaluated its impact on compile time and the execution time of optimized code. We have applied
our algorithm to applications that are an order of magnitude larger than those used in previous work. The experimental results demonstrate that interprocedural class analysis can significantly improve generated code quality, despite the imprecision of the information currently produced by our algorithm. We are working on improvements to the algorithm to eliminate the imprecisions introduced by parametric polymorphism, and expect that these enhancements will both reduce analysis time and increase the usefulness of the analysis as a tool for optimization.

Acknowledgments

This research is supported in part by an NSF Research Initiation Award (contract number CCR-9210990), an NSF Young Investigator Award (contract number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM Canada Ltd., and Pure Software. Craig Chambers and Jeffery Dean provided valuable comments on an earlier version of this paper. More information about the Cecil project is available via the World Wide Web at http://www.cs.washington.edu/research/projects/cecil.

About the Author

David Grove is a PhD candidate in the Department of Computer Science and Engineering at the University of Washington. His research focuses on the design and efficient implementation of advanced programming languages. He received his BS degree in computer science from Yale University in 1992, and his MS degree in computer science from the University of Washington in 1994. He can be reached at the Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350 USA, or via email at grove@cs.washington.edu.

References


