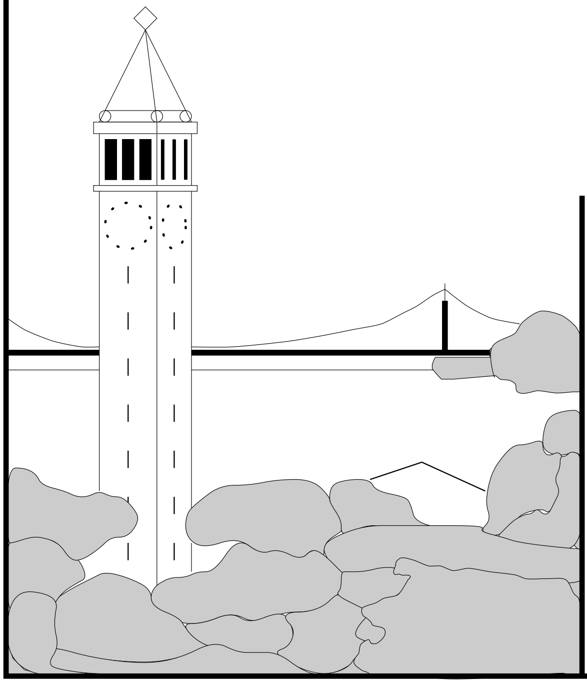
Fast and Effective Optimization of Statically Typed Object-Oriented Languages

David Francis Bacon



Report No. UCB/CSD-98-1017

December 1997

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Fast and Effective Optimization of Statically Typed Object-Oriented Languages

by

David Francis Bacon

A.B. (Columbia College) 1985

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Susan L. Graham, Chair
Professor Jonathan Arons
Professor Katherine Yelick

December 1997

Fast and Effective Optimization of Statically Typed Object-Oriented Languages

by

David Francis Bacon

Abstract

Fast and Effective Optimization
of Statically Typed Object-Oriented Languages

by

David Francis Bacon

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Susan L. Graham, Chair

In this dissertation, we show how a relatively simple and extremely fast interprocedural optimization algorithm can be used to optimize many of the expensive features of statically typed, object-oriented languages — in particular, C++ and Java.

We present a new program analysis algorithm, Rapid Type Analysis, and show that it is fast both in theory and in practice, and significantly out-performs other "fast" algorithms for virtual function call resolution.

We present optimization algorithms for the resolution of virtual function calls, conversion of virtual inheritance to direct inheritance, elimination of dynamic casts and dynamic type checks, and removal of object synchronization. These algorithms are all presented within a common framework that allows them to be driven by the information collected by Rapid Type Analysis, or by some other type analysis algorithm.

Collectively, the optimizations in this dissertation free the programmer from having to sacrifice modularity and extensibility for performance. Instead, the programmer can freely make use of the most powerful features of object-oriented programming, since the optimizer will remove unnecessary extensibility from the program.

Chair Date

To Laura,

for all your love and support

Contents

List of Figures

List of Tables

Acknowledgements

The faculty and students of the Computer Science Division at the University of California at Berkeley provided an incredible atmosphere of intellectual excitement and rigorous inquiry that will always stay with me. It was not only a privilege to work with so many people who loved their work, it was also an awful lot of fun.

My advisor Susan Graham gave me her support, advice, patience, encouragement, and friendship. Most of all she believed in me, which helped me believe in myself.

I was lucky to have Kathy Yelick and Jon Arons on my committee. They not only influenced my work, they also did so with such good humor that it was always a pleasure to work with them. I was also greatly influenced by Jim Demmel, who was always a gentleman and a scholar, in the most literal sense.

Oliver Sharp, Steve Lucco, Robert Wahbe, John Boyland, and Tim Wagner infected me with their enthusiasm, inspired me with the quality of their work, and taught me about "suesmanship".

At the IBM Watson Research Center, Shaula Alexander Yemini was first my colleague, then my manager, and finally my friend. Without her encouragement and her efforts to secure an IBM Resident Study fellowship for me, I might never have started this adventure. I was also fortunate to receive continued strong support from Danny Sabbah and Michael Burke, who "inherited" me after Shaula left IBM.

After my return to the Watson Research Center, Peter Sweeney contributed many ideas and a lot of hard work that made its way into this dissertation. Mark Wegman and Kenny Zadeck collaborated on the design of the Rapid Type Analysis algorithm. Harini Srinivasan, G. Ramalingam, Mike Hind, Mike Karasick, Mauricio Serrano, Shahani Weerawarana, and Michael Burke all made many contributions to this work – conversations, ideas, code, and written feedback.

Colleagues from around the world have also provided valuable feedback, data, benchmarks, and (to my chagrin) corrections – in particular, Yossi Gil, Craig Chambers, Urs Hölzle, Hemant Pande, and Brad Calder.

Other people who help me reach the finish line are Ben Fried and Andrew Mayer, who got me through some low points; Bill Hapworth, who helped me overcome my dissertation paranoia; the members of the Cal Sailing Club, who kept me sane; and Mark Kennedy, who goaded me until I swore he'd have to call me "doctor" one day.

My sister Sara and her partner Jennifer Balfour provided love, support, refuge and a sense of family three thousand miles from home. My mother, Gertrud, and my father, John, have always supported my education and gave me many gifts that went into this work.

Finally, my wife Laura Bennett gave me unwavering support, love, and humor.

Chapter 1

Introduction

1.1 Language and Optimization

Bjarne Stroustroup [1986] began his book on the C++ programming language with this quotation from the linguist B.L. Whorf:

Language shapes the way we think, and determines what we can think about.

Object-oriented programming is a real step forward in the expressive power of programming languages. A well-designed group of classes has a simplicity and conciseness that can not be achieved with the traditional procedural languages of the Algol family. Object-oriented languages help us to think about programming problems in a more modular, flexible manner.

But it is not only the language, but our experience of it, that shapes the way we think. In spoken language, if a phrase does not communicate our ideas effectively, we simply stop using it — first individually and eventually as a society. In a programming language, if a feature is inefficient, the feedback is less immediate — we often do not notice the inefficiency until we have already incorporated the feature into the critical portion of our program.

The experience of the Taligent Corporation can be seen as an extreme example. Founded by Apple and IBM to build an object-oriented operating system, Taligent's engineers quickly discovered that performance issues severely constrained their ability to build modular, re-usable software. The inefficiencies of object-oriented programming features played a significant part in the demise of the Taligent Corporation: hundreds of lost jobs, years of wasted effort.

As a result of such experiences, certain language features become linguistic archaisms. "Collective wisdom" dictates that programmers avoid such features, and while they continue to appear in the language specification, and are studied at universities, they disappear from common usage.

As programming languages become more and more high-level, optimization plays an increasing part in shaping the way we think. Optimization will determine which high-level features become part of everyday usage, and which features are relegated to obscurity.

1.2 Optimization of Object-Oriented Languages

In recent years, object-oriented languages have entered the mainstream. C++ and to a lesser extent Smalltalk are widely used in industry; Java is the lingua franca du jour of the Internet; and Modula-3, Eiffel, SELF, Dylan, and Cecil have gained a certain prominence in commercial and/or academic spheres.

The *pure* object-oriented languages (for instance Smalltalk and SELF) in which every operation, including integer addition, is a dynamically bound method invocation, are inherently inefficient unless significant optimization is performed. Therefore, several optimization techniques have been developed that are specific to the dynamic, untyped nature of these languages: method caching, message splitting, and so on.

Progress in optimizing "impure" languages (such as C++, Java, and Modula-3) has been slower for several reasons. First, the "impurity" is an attempt to solve efficiency problems in the language design, rather than in the optimizer, so there is less need for optimization and less room for dramatic improvements. Second, because of the relatively recent introduction and rapid acceptance of these languages, market pressures have been more strongly focused on features and functionality than on performance. Finally, in the case of C++, the complexity of the language led to a considerable delay before researchers had true compilers with which to study proposed optimizations. The combination of these factors has impeded efforts to develop sophisticated optimizations for C++ and Java and have left a significant performance gap.

This dissertation shrinks that gap.

1.3 My Thesis

The most expensive and most useful features of statically typed object-oriented programming languages like C++ and Java can be effectively optimized by a simple algorithm that is extremely fast.

1.3.1 An Optimization Framework

This dissertation describes a new analysis algorithm, Rapid Type Analysis, and a suite of optimizations that rely on the information it computes. The analysis computes a conservative set of live functions and live classes for an object-oriented program. Live functions are those that may be invoked during any execution of the program; live classes are those that may be instantiated during any execution of the program.

The live class and live function sets are used to drive optimizations that convert dynamic method dispatches into statically bound dispatches, that convert virtual inheritance to non-virtual inheritance, and that reduce or eliminate the cost of dynamic type inquiries and type casts.

Although we have demonstrated experimentally that our analysis algorithm embodies an excellent combination of accuracy and performance, many other analysis algorithms can be substituted as long as they compute the live procedure and live class sets in a conservative manner. Decoupling the analysis from the optimization is important because there is a broad spectrum of analysis algorithms, from class hierarchy analysis [Dean et al. 1995] to flow-sensitive and context-sensitive alias analysis [Pande and Ryder 1994]. The low end of the spectrum offers very fast execution at the cost of limited precision; the high end of the spectrum offers maximal precision at the cost of several orders of magnitude increase in compile time and space.

A major contribution of this dissertation is the development of a common framework for optimization of statically typed object-oriented languages, which allows different analysis algorithms to be "plugged in" to a single compiler. This in turn allows a more compact, modular implementation, reducing the cost of the compiler, and it allows the trade-offs between the various analysis algorithms to be computed in an "apples to apples" comparison.

1.3.2 Optimization by Specialization

The features that distinguish the object-oriented languages from other programming languages encourage modularity and reusability. Consequently, the biggest opportunities for new optimizations consist essentially of undoing modularity (by in-lining) and undoing re-usability (by specializing the code to a single application).

The best-known specialization is the conversion of dynamically bound (virtual, in C++ parlance) function calls into direct calls, thereby saving the dynamic dispatch overhead and enabling further in-lining. Virtual function resolution is a primary focus of this dissertation. We describe how our algorithm, Rapid Type Analysis (RTA), can be employed for virtual function resolution. We demonstrate through extensive measurements that RTA is almost as effective as expensive algorithms such as alias analysis, and is significantly more effective than simple algorithms such as class hierarchy analysis. Yet RTA is only marginally more complex to implement or costly in compile-time than class hierarchy analysis.

We also describe other specialization algorithms that we believe will have similarly excellent "price/performance" for conversion of virtual inheritance to non-virtual inheritance, and for compile-time implementation of run-time type identification (RTTI) features. We have not studied these optimizations experimentally because in C++ the features are either infrequently used (in the case of virtual inheritance) or not yet implemented in most compilers (in the case of RTTI). However, we expect that these features will become more commonplace over time due to their expressive power. RTTI is used extensively in Java.

The algorithms are all relatively simple and very fast. There is an unfortunate tendency in our field toward developing algorithms that are unnecessarily complex and therefore rarely used in practice. The depth of our work is not in the complexity of the algorithms, but rather in the judicious balancing of simplicity, effectiveness, and speed and in the experimental validation of our choices.

1.4 Scope of this Work

Where applicable, all algorithms are presented in this dissertation with variants for both C++ [Stroustrup 1986] and Java [Gosling et al. 1996]. Adaptation to other statically typed object-oriented languages should be relatively straightforward using either the C++

or the Java algorithm as a base. Such languages include Modula-3 [Harbison 1992], Eiffel [Meyer 1992], Dylan [Feinberg et al. 1997], Oberon-2 [Mossenbock and Wirth 1991] and Cecil [Chambers 1993], as well as the historically important languages Simula [Dahl and Nygaard 1966; Birtwistle et al. 1973; Nygaard and Dahl 1978], MAINSAIL [Wilcox et al. 1978], and Object Pascal [App 1988]

Some algorithms may be adaptable to the dynamic object-oriented languages (such as Smalltalk [Goldberg and Robson 1983] and SELF [Ungar and Smith 1987]), but the adaptation will be more complex and there may be better algorithms specifically targeted to such languages.

The Rapid Type Analysis algorithm and its dependent optimizations rely on having an entire program or library available for analysis, or on the intermediate representation including the necessary information (as described in Chapter 8). This means that these optimizations can not be applied directly to Java systems that dynamically load code. However, they can be applied to Java programs compiled in the traditional manner (`class` files contain sufficient information for RTA to analyze the program). There are already a number of such "static" Java compilers [Seshadri 1997; Proebsting et al. 1997; Hsieh et al. 1996; Cierniak and Li 1997; Bothner 1997].

All of the experimental work was done with C++. A colleague is implementing our algorithm for Java and already has some preliminary results [Serrano 1997].

Our algorithms and implementation all handle the various arcane features of the C++ language. In particular: virtual base classes, pointer-to-member functions, templates, and constructor semantics. This makes it possible for a compiler writer to use the algorithms directly, instead of having to adapt the algorithms and deal with the real-world complexities herself.

While the algorithms have broad relevance, the measurements may only be pertinent to C++. There are some significant differences between the object-oriented features of C++ and Modula-3, for example. Extrapolation from the C++ results to other languages should be done with care. Our benchmarking methodology was inspired by the SPEC benchmarks for C and FORTRAN [Dixit 1991; Dixit 1992], but C++ is a much more complicated language, and the programs vary more widely in their run-time characteristics. Therefore, benchmarking C++ effectively may require a much larger set of benchmark programs.

Optimizations serve two functions: to make existing programs smaller and faster, and to allow programmers to make more extensive use of high-level language features without paying a large performance penalty. While our measurements address the former, in the long run the latter effect is more important.

1.5 Contributions of this Dissertation

The contributions of this dissertation are:

- a fast, effective algorithm (Rapid Type Analysis) for computing the set of live procedures and live classes in an object-oriented program;
- a suite of optimizations that use the computed sets to resolve virtual function calls, de-virtualize inheritance, and convert dynamic type inquiries and type casts into compile-time expressions;
- an optimization framework that allows different analysis algorithms to be used with the optimization suite;
- measurements within that framework that compare the performance of several analysis algorithms for the purpose of virtual function call resolution, and that demonstrate the superiority of RTA;
- common data structures and concise notation to describe all the algorithms;

The Rapid Type Analysis (RTA) algorithm is currently scheduled for inclusion in an IBM product, and is being implemented by a number of other research groups in both academia and industry.

1.6 Contributors

Experimental computer science is by necessity a collaborative discipline, and much of the work described in this dissertation includes significant contributions from other people.

Peter Sweeney did most of the dynamic measurements and was a co-author of the paper that is incorporated in Chapter 5.

A variant of the Rapid Type Analysis algorithm described in Chapter 4 was developed independently and almost simultaneously by Mark Wegman and Kenny Zadeck; the final algorithm described here is a fusion of the best ideas from their algorithm and mine.

Harini Srinivasan did much of the design and implementation work for the Class Hierarchy Graph described in Chapter 3, and G. Ramalingam also contributed some important code.

1.7 Outline

Chapter 2 provides an overview of object-oriented program optimization, and places our work in context. Chapter 3 describes the data structures used by the analysis and optimization algorithms in the rest of the dissertation, and defines the terms and symbols used throughout. The Class Hierarchy Graph (CHG) is an important data structure and it must be understood to comprehend the RTA algorithm.

Chapter 4 describes the Rapid Type Analysis algorithm, which is used to provide the information to the optimizations described in the subsequent three chapters. However, other analysis algorithms can be used, and in this case it is only necessary to understand the analysis framework described in the first section.

Chapters 5, 6, and 7 describe the optimizations that can be performed once a program has been analyzed and dead classes have been identified: virtual function resolution, base class de-virtualization, and compile-time type identification.

Chapter 8 describes how the RTA algorithm can be applied when the complete program is not available to the optimizer, in particular when compiling and using libraries. Using RTA in a traditional separate compilation architecture is also discussed.

The work of the dissertation is summarized in Chapter 9, along with suggestions for future work in the area.

We have attempted to strike an appropriate balance between formality and complexity. To assist the reader, all variables, functions, and predicates used throughout this dissertation are defined in a *symbol glossary* in Appendix A.

Finally, Appendix B contains the data from the measurements that were used to produce the graphs in this dissertation.

Chapter 2

Related Work

This summary of related work will help to place our work in context, and provide a brief overview of the field of optimization of object-oriented programs. Readers familiar with the area may skip this chapter; more detailed discussions of related work will be included in the relevant sections.

2.1 Studies of Object-Oriented Programs

There have been a number of studies of the performance of object-oriented programs. Most of them have concentrated on statically typed languages, because in the pure dynamically typed languages all other costs are dominated by the dynamic dispatch overhead.

2.1.1 Studies of C++ Programs

For C++, Calder et al. [1994] investigated the differences between the C SPEC benchmarks and a set of their own C++ programs, and found that the C++ programs had shorter procedures, and performed more calls and indirect calls than the C programs. The C programs executed more conditional branches. The C++ programs issued more loads and stores, had worse instruction-cache locality, and allocated far more objects on the heap.

Driesen and Hölzle [1996] studied the cost of dynamic dispatch in eight C++ benchmarks and found that the overhead for dynamic dispatch was a median of 8% and a maximum of 18%. The overhead in terms of machine cycles was almost double the overhead in instructions, due to low instruction-level parallelism in dynamic dispatch code sequences.

They also investigated the effects of varying the branch mis-prediction penalty, branch target buffer size, load latency, and instruction issue width.

Wu and Wang [1996] studied the performance of a number of variations of a C++ implementation of the quicksort algorithm. The potential optimizations they identified were allocating small objects to registers, eliminating offset calculations in the absence of multiple inheritance (see Section 7.6), and static binding of types (in other words, type analysis).

2.2 Type Inference

Object-oriented programming derives much of its power from allowing one method to apply to many types. However, this is also one of the major sources of performance problems: because the type of the objects is not known, less optimization can be performed. The problem is most severe in dynamic, untyped languages such as Smalltalk, and least severe in statically typed languages such as C++.

Type inference is the process of inferring type declarations like those that would be present in a statically typed language by analyzing a program. Early work by Morris [1968] on the Lambda calculus and Reynolds [1969] on Lisp was made practical by Milner [1978] by using a fast unification algorithm. With a finite domain of types, a lattice-based data-flow technique can be applied [Jones and Muchnick 1976; Kaplan and Ullman 1980; Tennenbaum 1974].

Unfortunately, in the realm of object-oriented program analysis, the term *type inference* has been used in a number of different ways. Most important is the question of what is being used as the basis of the inferences. In dynamically typed languages, the basis for inference is method invocations: if the method `bar` is applied to object `foo`, then it is assumed that the dynamic type of `foo` is one of the classes that has a `bar` method.

However, `foo` might sometimes refer to an object without a `bar` method. Since many dynamically typed languages define a semantics for such a case (either a "no such method" method is invoked or an exception is raised) it is possible to write correctly functioning programs which contain type errors. In fact, the programmer may have written the code in such a way that he knows that the line of code invoking the `bar` method is never executed.

We use the term *type analysis* to refer to algorithms that use type declarations to provide the basic information about object types, and *type inference* to refer to algorithms that use method invocations as a basis.

Note that the results from a type inference algorithm can still be used to drive optimization of method invocations, as long as a dynamic test is included to verify the actual type.

2.2.1 Type Inference for Dynamic Languages

Suzuki [1981] developed an early type determination algorithm for Smalltalk-76 by extending Milner's algorithm.

Much of the recent work on type inference can be traced back to a constraint-based algorithm by Palsberg and Schwartzbach [1991] designed for a simple dynamic object-oriented language without aliasing. Their algorithm suffered from imprecision because each method was only analyzed once, leading to "pollution" of the type information in a caller by the types from other callers of the same method. The imprecision can be reduced by uniquely analyzing each method at each call site (effectively cloning the method for each call site). However, multiple levels of the call graph must be "cloned", leading to combinatorial explosion. The resulting algorithms are impractical.

Agesen et al. [1995] ameliorated this problem with an adaptive algorithm that only expands call chains that are likely to yield greater precision. Plevyak and Chien [1994] also developed an iterative algorithm with improved precision and speed.

Agesen [1995] improved on the precision of his previous work with his Cartesian Product Algorithm, which analyzes the Cartesian product of the possible argument types of a method. His algorithm was able to correctly infer the types of a `factorial` function but took nine seconds to do so, and is therefore impractical for real programs.

Palsberg and his colleagues have also continued to study and make refinements to their original algorithm [Schwartzbach 1991; Kozen et al. 1994; Palsberg and Schwartzbach 1994b; Palsberg and Schwartzbach 1995].

2.2.2 Type Analysis for Static Languages

Type analysis for statically typed object-oriented languages has developed from an entirely different background of interprocedural alias analysis.

The first such work was by Pande and Ryder [1994; 1995; 1996] for C++, based on the alias analysis algorithm of Landi and Ryder [1993] for C. Carini et al. [1995] also developed a type analysis algorithm for C++ based on that of Burke et al. [1994], and Steensgard [1996a] has developed a type analysis algorithm based on his almost-linear time algorithm for points-to analysis [Steensgaard 1996b].

2.2.3 Fast Algorithms

The information in statically typed programs makes it possible to do some type analysis by analyzing only the class hierarchy and call graph of the program. Calder and Grunwald [1994] first applied such an approach to C++ with their Unique Name algorithm, which resolves methods when their type signature is globally unique.

Fernandez [1995] and Dean et al. [1995] implemented versions of Class Hierarchy Analysis applied to Modula-3 and Cecil, respectively. Class Hierarchy Analysis resolves calls for which there is only one type-correct method in the class hierarchy, and is strictly more precise than Unique Name. Porat et al. [1996] compared Unique Name with profile-directed type feedback (see below) for C++ programs, and Diwan et al. [1996] compared Class Hierarchy Analysis with several intra- and inter-procedural flow-based analyses.

All of these algorithms are essentially constructing a call graph, and are therefore related to call graph construction algorithms for procedural languages [Ryder 1979; Südholt and Steigner 1992].

Our Rapid Type Analysis algorithm [Bacon and Sweeney 1996] is similar to these algorithms, and is strictly more precise than Class Hierarchy Analysis.

2.3 Eliminating Dynamic Dispatch

In addition to type analysis, there are two major techniques for reducing the frequency of dynamic dispatch: cloning and type prediction. Cloning generates multiple versions of methods that are specialized to parameter types, and type prediction inserts a fast test for the most likely type or types before defaulting to dynamic dispatch.

The SELF and Cecil projects have investigated these optimizations individually and together in the context of pure object-oriented languages [Ungar et al. 1992; Hölzle and Ungar 1994; Hölzle 1994; Chambers and Ungar 1991b; Chambers et al. 1991b; Dean et al. 1996].

2.3.1 Cloning

Cloning is the creation of duplicate function bodies specialized to the types or values of one or more arguments. A form of partial evaluation, particular types of cloning in an object-oriented context have been called *customization* and *specialization*.

Unrestricted cloning usually leads to an exponential increase in code size. Methods for addressing this problem have been put forward in the context of procedural language compilation [Cooper et al. 1993] and partial evaluation [Jones et al. 1993; Ruf and Weise 1991; Weise et al. 1991].

In object-oriented programs, the simplest type of cloning is customization: a specialized version of a method is compiled for each class that inherits it [Chambers and Ungar 1989a]. If the cloned method performs any self-dispatches, they can be statically bound because the type of the `self` or `this` pointer is statically known. For SELF, Chambers [1992] found that customization sped programs up by a factor of 1.5 to 5.

The term *specialization* has been used for cloning that may be based on the type of any of the parameters of a method, instead of just the receiver type as in customization. Dean [1994] has shown that specialization can improve performance of Cecil programs, and that increases in compile-time and code size can be limited effectively by using profile information to guide the specialization process [Dean et al. 1995].

Plevyak and Chien [1995] implemented a more general cloning algorithm for Concurrent Aggregates. Their technique is based on partitioning the program into equivalence classes by fixed-point iteration, and requires a modification to the method dispatch mechanism (because the types and values at the call site effectively become additional parameters). They tested their technique on ten small Concurrent Aggregates programs and found that the number of dynamic dispatches and the number of calls dropped considerably (there are fewer calls because of inlining). Code size increased as much as 70%, and no speedup figures were given.

Closely related to cloning is *message splitting* [Chambers and Ungar 1991a], which only replicates part of a method body rather than replicating the entire method. The purpose of message splitting is to reduce the loss of type information caused by merges in control flow, thereby allowing more methods to be directly dispatched.

2.3.2 Type Prediction

Type analysis and cloning both attempt to resolve method dispatches statically through program analysis. When this is impractical, either due to the cost of the static analysis or the unavailability of the whole program, significant optimization is still possible if the compiler can predict, by static analysis, profiling, or caching, the most likely class of the object.

Type prediction can also be used in conjunction with type analysis in dynamic environments (as for Java), where the class hierarchy may change in the future but is unlikely to do so. The types obtained from type analysis are used as predictions rather than certainties.

Static prediction was used in the SELF compiler [Chambers and Ungar 1989a]: when the receiver types of certain frequently used operators were not resolved by customization, they were predicted. For instance, the receiver of a "+" message was predicted as type integer and the receiver of an `ifTrue` message was predicted to be of type boolean.

Grove et al. [1995] studied profile-directed type feedback and found it to be highly effective. Agesen and Hölzle [1995] compared type inference to profile-directed type feedback, and found them to be similar in their run-time benefits with the exception that type inference often did not resolve calls to frequently used operators like "+".

A third mechanism for type prediction is inline caching [Hölzle et al. 1991]. When a method is dispatched dynamically, the target method is compiled as though it were being inlined into the context of the caller, and the call site is patched to jump to a stub that conditionally executes the inlined code based on the type of the object. Simple inline caches only keep a single inlined method body, and may thrash at heavily polymorphic call sites. Hölzle et al. show that it is almost always better to employ a *polymorphic inline cache*, which tests the alternatives in sequence. Code explosion and de-optimization can be limited effectively by bounding the number of inlined methods.

2.3.3 Designing Dynamics out of the Language

A variety of approaches have been taken in language design to reduce dynamic polymorphism and increase run-time efficiency. The statically typed languages such as C++ can be seen as an example of this approach. C++ forces the programmer to declare explicitly which methods will be dynamically dispatched (`virtual` functions) and which base classes will be found dynamically (`virtual` base classes).

The use of template-style mechanisms [Stroustrup 1989; Ellis and Stroustrup 1990; Lea 1990; Bank et al. 1996] is a further attempt to reduce dynamic binding and to improve compile-time checking. Often the code for a class, such as `List`, only exhibits *parametric polymorphism* rather than true polymorphism. That is, any type of object can be stored in a `List`, but all lists only contain one type of object. In this case, the parametric polymorphism can be declared by the programmer and the compiler will generate specialized versions of the classes based on their type parameters.

Templates are, in one sense, a mechanism that allows the programmer to perform cloning manually. They suffer similar problems of code expansion, most notoriously for container classes like `List`, where it is often unnecessary to generate separate method bodies for each type of pointer that the list can contain. However, templates have the advantage that they provide more static checking, because when an object is removed from a container class its type is statically known.

Trying to solve the same problem from the other direction, some researchers tried adding declarative type information to dynamic languages like Smalltalk. Ballard et al.'s QUICKTALK system implemented a type-checked subset of Smalltalk [Ballard et al. 1986]. The Hurricane [Atkinson 1986] and Typed Smalltalk [Johnson 1986; Johnson et al. 1988; Graver 1989; Graver and Johnson 1990] projects developed type declarations for Smalltalk and showed how they could be used by an optimizer to achieve substantial speedups. The Cecil language [Chambers 1993] continues in this tradition.

2.4 Reducing Code Size

Most of the work on type analysis has concentrated on using the computed type information for reducing dynamic dispatch. However, by constructing a call graph, it is also possible to remove unused methods. This is particularly important for the pure, dynamically typed languages like Smalltalk and SELF which include a large interactive programming environment, because when delivering a stand-alone application one does not want the entire programming environment gratuitously included in the executable.

Agesen and Ungar [1994] use type inference to address this problem in SELF by creating an application extractor. Srivastava developed a specialized algorithm similar to a call graph construction algorithm, which he used to identify and remove dead code from C++

programs.

Our Rapid Type Analysis algorithm, because it builds a call graph, can also remove unused functions.

Chapter 3

Data Structures

While statically typed object-oriented languages such as C++ provide programmers with a number of features that improve the flexibility and extensibility of their software, the overhead of implementing these features can be very expensive, as was discussed in Chapter 2. There is a lot of on-going and earlier work in the area of compiler optimization of object-oriented programs, aimed at improving the performance of these programs [Carini et al. 1995; Pande and Ryder 1994; Chambers et al. 1991b; Chambers and Ungar 1991a; Agesen and Hölzle 1995; Hölzle et al. 1991; Plevyak and Chien 1994; Diwan et al. 1996]. While these papers present interesting new ideas and algorithms for optimizing object-oriented programs, there has been very little focus on improving the performance of these optimization algorithms and very little mention of the algorithms to build the data structures used to implement these optimization algorithms.

Traditionally, for imperative programs such as C and Fortran, compiler optimization algorithms have been developed using very efficient intermediate representations of the source code. Some of these representations have found their way as data structures in several commercial compilers. The Static Single Assignment form [Cytron et al. 1991], for example, has proven to be a very powerful platform for several code optimizations implemented in most modern compilers for C and Fortran. These intermediate representations primarily represent the control flow and data flow information in imperative programs. While some of these intermediate representations can be effective in compilers for object-oriented languages, they do not represent the information on object-oriented features available in these languages. The focus of the work presented in this dissertation has been development of efficient intermediate representations (data structures) to represent such features. These intermediate representations can serve as efficient platforms for several object-oriented analyses and optimizations proposed in the literature.

One of the most important data structures necessary to optimize any object-oriented program is a graph that represents the class hierarchy and the method inheritance relationships in the program. This chapter describes such a data structure, the Class Hierarchy Graph (CHG), and a fast algorithm to compute it. One of the most important uses of the CHG is to compute the Program Virtual-call Graph (PVG). The PVG is a variant of the traditional call graph used in compilers for C and Fortran, extended to include C++ features such as virtual functions, pointer-to-member functions, etc. We have found the CHG and the PVG to be fundamental data structures for a variety of optimizations of object-oriented language features.

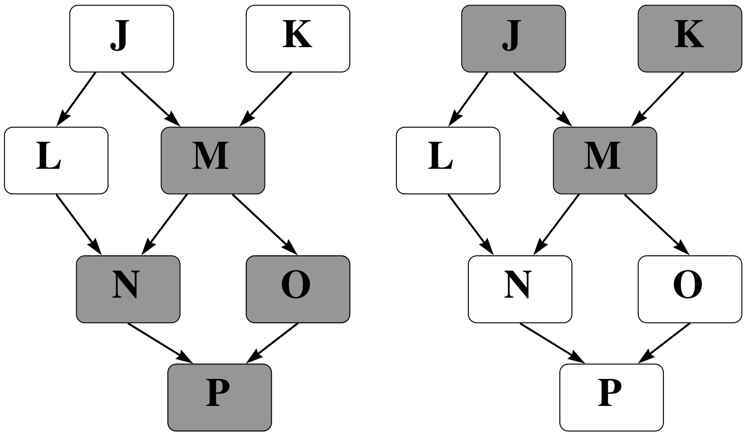
Set-theoretic notation will be used to present the data structures, so that the algorithms can be described concisely. The judicious linking of objects by pointers removes the need for most of the expensive-seeming operations in the actual implementation.

3.1 The Class Hierarchy Graph

In statically typed object-oriented languages, a class `D` can be derived from another class `B`; `B` is called the base of the derived class, `D`. A derived class inherits the properties of the base class, including the data members and member functions. The derived class can also *override* the functions of its bases. We use the Class Hierarchy Graph (CHG) to represent the inheritance relationships between classes. The Class Hierarchy Graph also represents the various member functions (including virtual functions) inherited, overridden or defined in each class. In this section, we present a formal definition of the CHG and illustrate the CHG using an example. In Section 3.1.1, we describe algorithms to build the CHG. Section 3.2 discusses the application of the CHG to computing sets of inherited and overridden methods, which are crucial for subsequent analyses.

A *Class Hierarchy Graph* is a tuple $< C, D, V >$ where

- C is the set of *classes*, which form the nodes of the graph,
- D is the set of *derivations*, which are ordered pairs of classes forming the edges of the graph,
- V is the set of *visible methods*, which are tuples that represent the methods that can be invoked through a reference to an object of a particular class type.



(a) *Derived**(M) (b) *Bases**(M)

Figure 3.1: Illustration of $Derived^*$ and $Bases^*$ with respect to class M in the CHG. Shading denotes that the class is part of the set.

A derivation edge in D is an ordered pair $< b, d >$ where $b \in C$ is the base class and $d \in C$ is the derived class. Thus, the set of class nodes and the set of derivation edges represent the inheritance relationships in the class hierarchy. Note that unlike the standard C++ representation, in our representation, inheritance arcs, *i.e.*, derivation edges, flow from the base class to the derived class. This is because most traversals during program analysis and optimizations go down the class hierarchy (traversing the base classes before the derived classes), rather than up[1]. In the presence of multiple inheritance, a class node can have multiple immediate predecessors (parents) in the CHG. We refer to the in-degree of a class node in a CHG as the *degree of inheritance*. Class nodes whose degree of inheritance is zero are referred to as *root* nodes.

The set of base classes of a class c is denoted $Bases(c)$. Formally, $Bases(c) = \{b \in C : < b, c > \in D\}$. Similarly, the set of classes derived from class c is denoted $Derived(c)$, where $Derived(c) = \{d \in C : < c, d > \in D\}$.

We frequently make use of the set of transitively derived classes or transitively inherited classes of a class. We define $Derived^*(c)$ as the set of class nodes $d \in C$, including c itself, such that there is a path from c to d in the CHG. Similarly, we define $Bases^*(c)$ as the set

[1]To facilitate traversals in either direction, in our implementation, we have bidirectional edges between nodes in the CHG.

of class nodes $d \in C$, including c itself, such that there is a path from d to c in the CHG. These concepts are illustrated in Figure 3.1.

Let M_D be the set of all methods with code bodies defined in a program; let M_I be the set of all method interfaces introduced in a program (in C++ terms, the *pure virtual* methods; in Java terms, the *abstract* methods). The set of all methods $M = M_D \cup M_I$ includes both concrete methods with code bodies as well as method interfaces without code bodies.

The set of visible methods, V, is the set of methods and interfaces that a class declares or inherits, for all classes $c \in C$. The visible methods are computed according to the particular inheritance and override semantics of the source language. Formally, a visible method $v \in V$ is a tuple $< c, m, d >$ where

- $c \in C$ is the class in which v represents a visible method,
- $m \in M$ is the method which is declared or inherited by c, and
- $d \in C$ is the defining class of m. Either $c = d$ (the method is declared by the class, not inherited), or $c \in \mathit{Derived}^*(d)$ (the method is inherited from a base class).

The set $V_c = \{m \in M, d \in D : < c, m, d > \in V\}$ represents the set of methods that are defined for class c. This includes both the actual methods with code bodies $m \in M_D$ that c defines or inherits, as well as the interfaces $n \in M_I$ that c defines or inherits. In the next section we will show how the set V is constructed from the input program.

Example Consider the C++ program in Figure 3.2. The CHG for this program is given in Figure 3.3. Nodes `w`, `x`, `y`, `z` and `u` represent class nodes. Within each class node are the visible method subnodes corresponding to each method visible in that class. For example, methods `x::g` and `z::f` are visible in class node `z`. The next subsection presents an algorithm to construct these visible method subnodes.

For readers unfamiliar with C++, the use of *virtual* inheritance along CHG edges $< w, y >$ and $< w, x >$ in the above example indicates that objects of type u should only contain a single sub-object of type w, instead of one for each of the two paths along which it was inherited.

```
class w {
   virtual void g() { ... };
};
class x : public virtual w {
   virtual void g() { ... };
};
class y: public virtual w {
};
class z : public x {
   virtual void f() { ... };
};
class u : public y, public z {
   void f() { ... };
};
```

Figure 3.2: A group of C++ class declarations

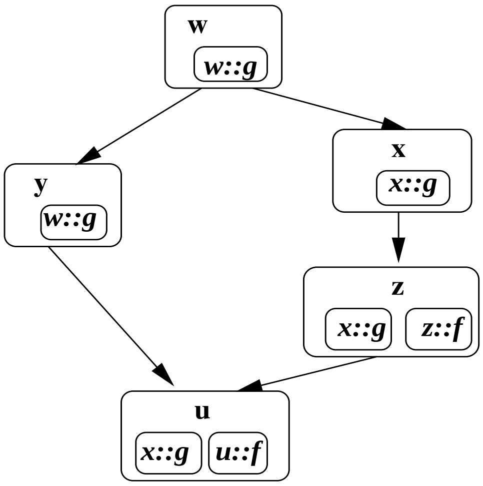


Figure 3.3: Class Hierarchy Graph of Program in Figure 3.2. Edges $< w, y >$ and $< w, x >$ are virtual inheritance edges.

3.1.1 Building the CHG

The class nodes and derivation edges of the CHG for a C++ program are constructed from source-level information, *i.e.*, from the class declarations and the inheritance relationships specified in the program. In this section, we present an algorithm (Figure 3.4) to compute the set of methods *visible* at any class node, given the methods *declared* in each class node. Since any method visible in a base class node is also visible in a derived class node, the algorithm in Figure 3.4 processes nodes in the CHG in topological order. Algorithm **assignTopologicalNumbers(C)** assigns topological numbers, *TopNum*, to the class nodes in C such that the *TopNum* of a node is always greater than the *TopNum* of all its parent nodes, and the *TopNum* of an implementation class is always greater than the *TopNum* of an interface class that is its sibling. Topological numbers are assigned sequentially beginning with 1, and each class has a unique topological number.

At a root node the set of visible methods is simply the set of methods declared in the node. At all other nodes, the set of visible methods is the union of the methods declared in the current node and the methods visible in its parent nodes.

Since the CHG can potentially be a directed acyclic graph, we have to consider multiple paths along which different methods with the same signature may be declared. For example, in Figure 3.3, method `g()` is defined in nodes `w` and `x`. If while processing node `u`, in line 9 of Algorithm **buildCHG**, derivation edge $< \mathtt{y}, \mathtt{u} >$ is considered before edge $< \mathtt{z}, \mathtt{u} >$, the visible method $< \mathtt{u}, \mathtt{w} :: \mathtt{g}, \mathtt{w} >$ is added to V. However, the method reaching `u` is `x::g`. Lines 10-13 of Algorithm **buildCHG** uses the signature information to handle this case. The *signature* of a method m, $Sig(m)$, is essentially the name of the method and its type. If the signatures of a method visible in a parent node and a method computed to be visible in the child node match, we use the topological number of the defining classes in the two visible methods to determine which of the two methods is visible in the child node. If in line 11, $TopNum(d) > TopNum(e)$, then, the method defined in d overrides the method in e. Hence, the method visible in class c must be the method defined in d. Using this technique, for our example CHG, the visible method set, V, is updated as follows: since $TopNum(\mathtt{x}) > TopNum(\mathtt{w})$, $< \mathtt{u}, \mathtt{w} :: \mathtt{g}, \mathtt{w} >$ is removed from the list and $< \mathtt{u}, \mathtt{x} :: \mathtt{g}, \mathtt{x} >$ is added instead.

What about the case when two base classes `a` and `b` of a class `c` both define a method with the same signature, but neither one is a base of the other? In C++, class `c` would be ambiguous and therefore illegal unless it explicitly overrode the method. In Java, either

```
buildCHG(C, D, M)
    V ← ∅
    assignTopologicalNumbers(D)
    for each m ∈ M
        V ← V ∪ {< ClassOf(m), m, ClassOf(m) >}
    for i = 1 to |C|
        Let c ← x ∈ C : TopNum(x) = i
        for each b ∈ C : < b, c >∈ D
            for each m ∈ M, d ∈ C : < b, m, d >∈ V and not Private(m)
                if ∃_{n∈M,e∈C} : < c, n, e >∈ V and Sig(m) = Sig(n)
                    if TopNum(d) > TopNum(e)
                        V ← V − {< c, n, e >}
                        V ← V ∪ {< c, m, d >}
                else
                    V ← V ∪ {< c, m, d >}
```

Figure 3.4: Algorithm to Construct the Class Hierarchy Graph

both methods are interfaces and it does not matter which one is chosen, or else one is a method implementation and the other is an interface (since Java does not have multiple inheritance). In this case, the implementation method will be chosen because the interface method will have the lower topological number.

3.1.2 Complexity

In this section we analyze the worst-case and expected-time complexity of the CHG construction algorithm. Throughout this dissertation we will abuse notational convention by writing $O(X)$ for $O(|X|)$.

For worst-case complexity bounds we will assume that set and mapping data structures are implemented with a data structure that allows insert and delete operations to be performed in $O(\log n)$ time, where n is the size of the set. In our expected-time complexity bounds (as in our actual implementation) we assume that sets and mappings are implemented with hash tables that have $O(1)$ expected time for insert and delete operations, but a worst case time of $O(n)$.

Building the CHG consists of two steps: assigning the topological numbering, and adding the visible function entries. The topological numbering is created in $O(C + D)$ worst-case time using a standard work-list algorithm.

The **if** clause in the innermost loop compares function signatures. This can be done in constant time if function signatures are represented by unique numbers. In our compiler, unique numbers are generated for each function signature, and these numbers are used even when generating non-optimized code, so we do not count the cost in our total complexity. However, the cost of building the set of unique signatures is $O(M \log M)$ worst-case time.

The other operations performed in the inner loop are insertions and removals into the set V. In the implementation, the set V is implemented as disjoint sets residing at each class node. That is, the node for class c contains the subset $V_c = \{m \in M, d \in D : < c, m, d > \in V\}$. There are at most three lookup/remove/insert operations in the inner loop, and each operation will cost $O(\log V_c)$ worst-case time.

Thus the total worst-case cost of lines 10–15 in Figure 3.4 is $O(\log V_c)$, since the set lookup, removal, and insertion can each be performed in that amount of time.

Now we must account for the loops. The loop on line 9 iterates over the functions of the base classes of c. For any base class b of class c, every function in its visible function set V_b must be either inherited or overridden in V_c. Thus $|V_c|$ is an upper bound on the number of iterations by the loop on line 9. The number of iterations by the loop on line 8 is the number of classes from which c inherits, that is, the degree of inheritance. We denote this quantity as $|Bases(c)|$. The total number of steps in the algorithm can now be expressed as

$$O\left(\sum_{c \in C} Bases(c) V_c \log V_c\right)$$

We now observe that if we use the maximum values of the number of base classes

$$\mathcal{B} = \max_{c \in C} |Bases(c)|$$

and the number of visible methods for a class

$$\mathcal{M} = \max_{c \in C} |V_c|$$

then we can rewrite the worst-case running-time equation as

$$O\left(\sum_{c \in C} \mathcal{B}\mathcal{M} \log \mathcal{M}\right)$$

which is equivalent to

$$O(C\mathcal{B}\mathcal{M} \log \mathcal{M}).$$

Expected Complexity

All of the complexity bounds in this dissertation suffer from a common problem: worst case numbers assume that the class hierarchy can be an arbitrary DAG, with classes inheriting thousands of other classes, defining millions of methods, etc. In practice, the fact that the class hierarchy is structured to conform to the functional relationships of objects means that such class hierarchies will never exist in practice.

Unfortunately, it is difficult to capture this observation formally. Therefore, it is important to keep track of which complexity terms are bounded in practice so that attempts at optimization are directed at those complexity terms which really would grow with the size of the input.

The multiplier $log\,\mathcal{M}$ accounts for the set lookup operations in the inner loop. However, if the sets V_c in each class are implemented with a hash table, then the expected time to perform a lookup will be constant.

Practically speaking, the degree of multiple inheritance $\mathcal{B}$ is almost never larger than 4, since this would imply an extremely complex relationship between objects which would be confusing to programmers. Therefore, $\mathcal{B}$ can essentially be regarded as a small constant.

As a result of these two simplifications, the expected-case running time of the algorithm is improved to

$$O(C\mathcal{M}).$$

While the maximum number of methods $\mathcal{M}$ visible at a class is in some real cases in the hundreds, the average number of visible methods at a class is usually not very large, as we will show in Section 3.3.

Finally, it should be noted that the visible method information must be computed to compile the method dispatches in the program, so in this sense the CHG construction does not impose any additional cost.

3.1.3 Adapting the CHG for Java

While Java only has single inheritance of implementation, it does have multiple inheritance of interfaces. It is therefore not significantly different from C++ in terms of the complexity of the class hierarchy.

The only adjustments necessary to the algorithm have to do with the peculiar way in which Java defines the inheritance semantics of the `Object` class. Any implementation class which does not extend some other implementation class implicitly extends the `Object` class. On the other hand, interface classes do not share a common root class. Nevertheless, any of the methods of the `Object` class can be invoked on a reference to an interface type. This is justified by the fact that every interface must point to an instance derived from class `Object`.

It is unclear why the designers of Java chose this inelegant solution when they could have simply created an `ObjectInterface` type which roots the interface hierarchy, and then have `Object` be derived from `ObjectInterface`. One possible reason is that not all methods of `Object` are public, and all methods of interface classes are implicitly public.

The way to deal with this peculiarity is simply to have `Object` be an implicit base class of all interface classes that do not extend some other interface. Once this is done, the visible method information will be correctly propagated through the CHG. There are indications that the language designers sometimes use this approach themselves: the `javadoc` tool, which automatically generates class documentation, shows all interfaces as being derived from class `Object`.

3.2 The Override Frontier

Statically typed object-oriented languages allow a method in class c to be overriden by another method of the same signature in a derived class d. The set of classes that override a particular class's method, the *Override set*, and the set of classes that inherit a particular class's method, the *Inherit set*, are used in a number of analysis and optimization algorithms.

Conceptually, for a particular visible method, $v =< c, m, d >$, the Inherit and Override sets determine a frontier (boundary) in the CHG for the visible method. Such a frontier separates the nodes in the CHG that inherit the method m from the nodes in the CHG that inherit or define a different method $n \neq m$ such that $Sig(m) = Sig(n)$. We refer to this frontier as the *override frontier*.

The algorithm **buildFrontier** is essentially propagating back up the class hierarchy information about what the **buildCHG** algorithm propagated down the class hierarchy. The result is a kind of sparse evaluation graph for inheritance relationships: the override set of a visible method "points" to all of the classes in which it is overridden. Since the visible method represents the static type of a virtual function call site, recursively visiting

Visible Method	Override	Inherit
$< \mathtt{w}, \mathtt{w} :: \mathtt{g}, \mathtt{w} >$	{x}	{w,y}
$< \mathtt{y}, \mathtt{w} :: \mathtt{g}, \mathtt{w} >$	{u}	{y}
$< \mathtt{x}, \mathtt{x} :: \mathtt{g}, \mathtt{x} >$	$\emptyset$	{x,z,u}
$< \mathtt{z}, \mathtt{x} :: \mathtt{g}, \mathtt{x} >$	$\emptyset$	{z,u}
$< \mathtt{z}, \mathtt{z} :: \mathtt{f}, \mathtt{z} >$	{u}	{z}
$< \mathtt{u}, \mathtt{x} :: \mathtt{g}, \mathtt{x} >$	$\emptyset$	{u}
$< \mathtt{u}, \mathtt{u} :: \mathtt{f}, \mathtt{u} >$	$\emptyset$	{u}

Figure 3.5: Visible Methods for Example Program

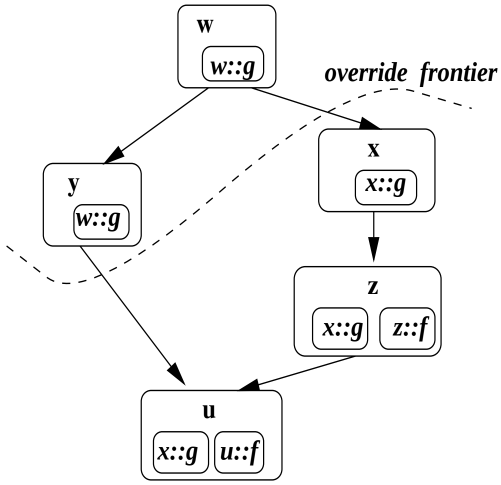


Figure 3.6: Class Hierarchy Graph showing Override Frontier of `w::g`

its override sets yields the set of methods that could be dynamically invoked at that call site.

While the *Override* set helps to find possible *methods*, the *Inherit* set helps to find possible *classes*. Assume that we have already used the *Override* sets to find the set of methods that could be invoked at a virtual call site. Then the set of classes through which each method could be invoked is given by the *Inherit* set at each of those visible methods.

Consider our example program, whose CHG is shown in Figure 3.3: the visible method subnodes are expanded in the table in Figure 3.5. For visible method $< \mathtt{w}, \mathtt{w} :: \mathtt{g}, \mathtt{w} >$ in class node `w`, the Override set is {`x`} and the Inherit set is {`w`,`y`}. The override frontier is given in Figure 3.6. Class nodes `w` and `y` on one side of the frontier inherit the method `w::g`. The nodes on the other side of the frontier, `x`, `y` and `u`, instead define or inherit a different implementation of `g()`, namely `x::g`.

The Inherit and Override sets are used in the *Program Virtual Call Graph* construction algorithm described in Section 3.4. The Inherit and Override sets can also be used to resolve virtual function calls to direct function calls. For example, in our example program, class `x` defines function `g()`. Since none of the descendants of `x` in the CHG override `g()`, any virtual function call of the form, $p \to g()$, where the static type of the object that p points to is `x` can be resolved to `x::g`.

Computing Override and Inherit Sets

The computation of the Override and Inherit sets proceeds in reverse topological order, *i.e.*, visiting derived class nodes (child nodes) before base class nodes (parent nodes) in the CHG. The algorithm to compute the Inherit set (*Inherit*) and the Override set (*Override*) for all visible method subnodes in the CHG is shown in Figure 3.7. Since there are no descendants for a leaf node, any method visible in a leaf node, l, can not be overridden. Likewise, there are no descendants of l that can inherit any of its visible methods. Therefore, for all visible methods in a leaf node, l, the *Inherit* set is $\{l\}$ and the *Override* set is $\emptyset$. Lines 2-5 initialize the *Inherit* and *Override* sets for all visible methods.

Line 6 initializes the *Antiset* of each visible method. The *Antiset* is required for class hierarchies in which a method is overridden by inheritance rather than by definition, as is the case with the method `g()` in Figure 3.6. At class `u`, `w::g()` is overridden by `x::g()`, which is inherited. However, class `u` does not belong in *Override*(w), because it already contains `x`, which accounts for the override by method `x::g()` (see Figure 3.5). Essentially, the *Antiset* is used to detect and remove redundant override entries caused by class hierarchies with multiple paths between classes. The *Antiset* is a temporary data structure that is not used beyond the frontier computation.

The loop on line 7 processes class nodes in reverse topological order (from the leaves to the roots). For each node c processed in this loop, we examine each of the visible methods of class c (line 9). We visit each of the parent classes b of class node c in the CHG (line 11), and update their *Inherit* and *Override* sets (lines 12–25). At a parent node, b, *Inherit*(w) and *Override*(w) sets for a visible method, w, are computed based on the corresponding sets in its child node c. To understand the algorithm, the diagram in Figure 3.8 may be helpful.

Line 12 checks whether the base class b has a visible method with the same signature

```
buildFrontier(C, D, V)
    for each v ∈ V
        Let < c, m, d >= v
        Override(v) ← ∅
        Inherit(v) ← {c}
        Antiset(v) ← ∅

    for i = |C| to 1 step -1
        Let c ← x ∈ C : TopNum(x) = i
        for each m ∈ M,  d ∈ C : < c, m, d >∈ V
            Let v =< c, m, d >
            for each b ∈ C : < b, c >∈ D
                if ∃_{n∈M, e∈C} : < b, n, e >∈ V and Sig(m) = Sig(n)
                    Let w =< b, n, e >
                    if d = e
                        Inherit(w) ← Inherit(w) ∪ Inherit(v)
                        Override(w) ← Override(w) ∪ Override(v)
                    else
                        Override(w) ← Override(w) ∪ {v}
                        if c ≠ d
                            for each p ∈ C : < p, c >∈ D and < p, m, d >∈ V
                                Antiset(p) ← Antiset(p) ∪ {v}
                    Antiset(w) ← Antiset(w) ∪ Antiset(v)
                    q = Override(w) ∩ Antiset(w)
                    Override(w) ← Override(w) − q
                    Antiset(w) ← Antiset(w) − q
```

Figure 3.7: Algorithm to Compute Inherit and Override Sets

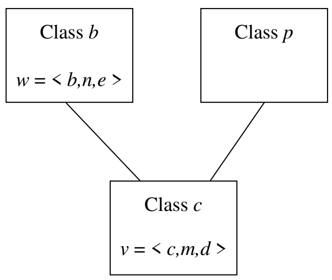


Figure 3.8: Illustration of Frontier Algorithm

as visible method v of class c. If there is not, then we simply move on to the next base class. If there is, the visible method of base class b is denoted by w.

Line 14 of the algorithm checks whether the defining classes (d and e) of the two visible methods are the same. If they are, then class c is inheriting v from class b. In that event, the contents of the *Inherit* and *Override* sets of class c are simply added to the corresponding sets in the base class b (lines 15 and 16).

If $d \neq e$, then visible method v in class c is overriding visible method w of base class b. In that event, the *Override* set of b is updated to include v, but no *Inherit* information is propagated to b (line 18).

The rest of the algorithm (lines 19–25) is only required to handle "sibling overrides", which do not occur very often in real programs (we were unable to find any real programs that made use of this feature). However, sibling overrides can occur, as in Figure 3.6 where the `g()` method defined by class `x` overrides the `g()` method of its sibling `y` at class `u`.

In the event that there are sibling overrides, there will be override entries in the siblings that should *not* be propagated to their parents. The *Antiset* at each visible method is used to handle this case.

If $c \neq d$, then visible method v has been inherited from some sibling of class b and is overriding visible method w (line 19). In that event, v is added to the *Antiset* of each base class p of c from which the overriding method was inherited (lines 20 and 21).

Lines 22–25 propagate the *Antiset* to the base class b, whether v was inherited or overridden. Any visible methods that were in the *Antiset* of class c are removed from both the *Antiset* and the *Override* set of the base class b.

Complexity

We now present the time complexity of Algorithm **buildFrontier**. The initialization phase, lines 3–6, requires constant time for each visible method. Hence, the complexity of the loop in line 2 is $O(V)$.

Since they are sets of classes, the maximum size of each *Antiset*, *Inherit*, and *Override* set is C. The insertion of a single element on lines 18 and 21 therefore takes $O(\log C)$ time. The set union, intersection, and difference operations on lines 15–16 and 22-25 could each require as many as C insert, lookup, or delete operations, and therefore require $O(C \log C)$ time.

The loop on line 20 iterates $Bases(c)$ times, so the total cost of the loop of lines 20–21 is $O(Bases(c)\log C)$. Line 12, which looks up the method m in the base class, costs $O(\log V_b)$ which can be bounded by $O(\log V_c)$.

Therefore, the total cost of the loop body of lines 12–25 is

$$O(\log V_c + Bases(c)\log C + C\log C).$$

We also observe that $Bases(c)$ is always dominated by C, so the middle term can be dropped. The loops on lines 7 and 9 together iterate over all of the visible methods, and the loop on line 11 iterates $Bases(c)$ times. Therefore, the complexity of the entire algorithms can be expressed as:

$$O\left(\sum_{<c,m,d>\in V} Bases(c)(\log V_c + C\log C)\right)$$

We have defined $\mathcal{B}$ and $\mathcal{M}$ be the maximum values over all classes $c \in C$ for $|Bases(c)|$ and $|V_c|$ respectively. Therefore, the time complexity is:

$$O(V\mathcal{B}(\log\mathcal{M} + C\log C)).$$

Expected Complexity

If the Override and Inherit sets are represented as hash tables, the logarithmic factors for the lookups can be dropped.

The factor $\mathcal{B}$ represents the degree of inheritance of a class node. As mentioned earlier, $\mathcal{B}$ can be bounded by a constant in practice since multiple inheritance is rare and is usually at most 4. Therefore the expected time of the algorithm is

$$O(VC).$$

The multiplier C is due to the fact that Override or Inherit sets could include all (or almost all) classes. In practice, these sets usually only contain a small number of members. Only when there is a common root class (as with Java's `Object` class) will the sets approach this size, and then only for the methods defined by `Object`, of which there are a small number.

Therefore, in practical terms we expect the algorithm to be linear in the size of V.

Program	C	$\mathcal{B}$	V_c		*Override*		*Inherit*		*Antiset*	Connected
			$\mathcal{M}$	Avg	Max	Avg	Max	Avg	Max	
sched	58	3	142	26.4	3	1.3	5	1.4	0	11
ixx	95	1	93	21.0	27	2.2	32	1.7	0	35
lcom	85	2	139	30.3	24	3.4	33	1.7	0	33
hotwire	37	2	60	13.1	11	2.7	17	2.1	0	17
simulate	55	2	140	27.9	5	2.2	9	1.5	0	10
idl	84	2	139	40.9	8	1.8	16	2.1	0	44
taldict	39	2	163	22.3	2	1.1	4	1.2	0	4
deltablue	10	1	22	19.3	5	5.0	1	1.0	0	6
richards	12	1	20	13.7	4	4.0	1	1.0	0	6

Key:

C	Number of Classes	$\mathcal{B}$	Maximum base classes
V_c	Visible methods per class	$\mathcal{M}$	Maximum V_c
Override	*Override* set size	*Inherit*	*Inherit* set size
Antiset	*Antiset* size	Connected	Largest connected subset of C

Table 3.1: Static Properties of the Class Hierarchy

3.3 Program Properties

The expected-time complexity arguments we have made depend partly on the presumption that the complexity of the class hierarchy will naturally be restricted due to software engineering principles and the inability of programmers to conceptualize very complex class structures. In this section we present some measurements to support these claims.

The properties of a number of benchmark programs are shown in Table 3.1. The benchmarks are the same ones used to evaluate the use of Rapid Type Analysis for program optimization, and are described fully in Section 5.4.2. The first seven benchmarks are real programs of medium size; the last two are commonly used small benchmarks that are provided for comparison purposes.

Most surprising is that the total number of classes used by the programs is quite small (never more than 100), even with the inclusion of `iostream` and other library classes. While the number of classes would be expected to grow quite considerably with truly large programs (1 million lines of code or more), it may be that the number of classes will not be as large as has been generally supposed.

The set of visible methods at class c, V_c, has an average size ranging from 13 to 41.

```
void main() {
  w* wp = new w;
  y* yp = new y;
  wp->g();
}
```

Figure 3.9: Code of the example program

Surprisingly, more than half of the programs had classes with more than 100 visible methods. While the maximum number of visible methods per class ($\mathcal{M}$) varies considerably across programs, the average varies considerably less and remains within the bounds that we would expect based on software engineering principles. Therefore, as a rule of thumb we can expect both the time and space cost of the CHG to be proportional to about 20 times the number of classes.

The largest *Override* and *Inherit* sets are close or equal in size to the largest connected subset of the class graph, showing that there do exist some quite complex patterns of method inheritance. However, the average size of both sets never exceeds 4 for real programs.

The *Antiset* sets are all empty, indicating that there are no methods overridden by sibling method definitions. The lack of sibling overrides is not surprising, since multiple inheritance is rare in the first place and sibling overrides are questionable practice from a software-engineering standpoint.

Therefore, we really can expect the **buildFrontier** algorithm to take time proportional to the size of V, the set of visible methods.

3.4 The Program Virtual Call Graph

The analysis required for our optimizations is carried out using a program call graph extended to take the semantics of virtual function calls into account (called the Program Virtual-call Graph, or PVG), and a Class Hierarchy Graph (CHG) which represents the class hierarchy and method inheritance relationships, and is designed to allow the PVG construction and the analysis to be performed rapidly. We have found these two data structures to be fundamental for a wide variety of object-oriented optimizations.

We will illustrate the PVG using the class hierarchy from Figure 3.2 whose CHG is

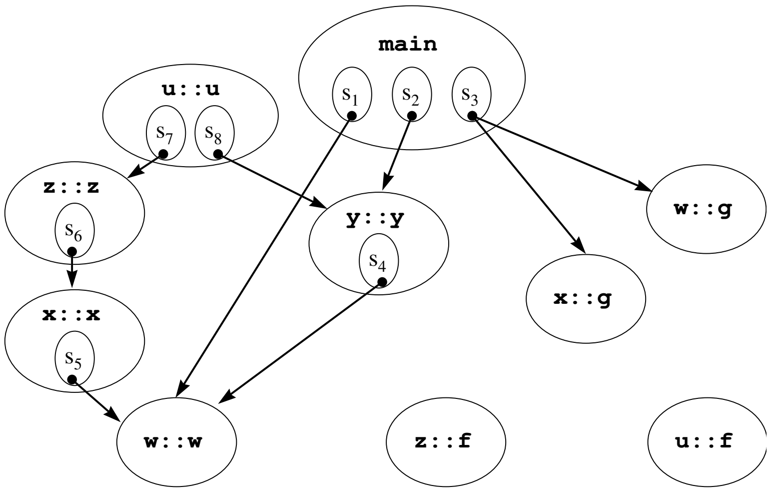


Figure 3.10: PVG of the example program in Figure 3.9.

shown in Figure 3.3. Figure 3.9 is an example program using those class declarations. It simply creates objects of types `w` and `y`, and then invokes the virtual function `g()` via the pointer, `wp`, to the `w` object.

3.4.1 Formal Description

The PVG is a call graph, extended to handle the semantics of virtual function calls. The PVG is a tuple $< F, S, I, R >$ where

- F is the set of *function* nodes (F includes methods and non-methods, so the set M_D of methods with code bodies is a subset of F),
- S is the set of *call site* sub-nodes,
- I is the set of *call instance* edges, and
- $R \subseteq F$ is the set of *roots* of the call graph.

The PVG for our example program is shown in Figure 3.10. The set of methods is

$$M_D = \{\mathtt{w} :: \mathtt{w}, \mathtt{y} :: \mathtt{y}, \mathtt{x} :: \mathtt{x}, \mathtt{z} :: \mathtt{z}, \mathtt{u} :: \mathtt{u}, \mathtt{w} :: \mathtt{g}, \mathtt{x} :: \mathtt{g}, \mathtt{z} :: \mathtt{f}, \mathtt{u} :: \mathtt{f}\}$$

the set of functions is

$$F = \{\mathtt{main}\} \cup M_D$$

$$
\begin{array}{llllll}
< & s_1, & \texttt{main}, & \texttt{w}::\texttt{w}, & \perp & > \\
< & s_2, & \texttt{main}, & \texttt{y}::\texttt{y}, & \perp & > \\
< & s_3, & \texttt{main}, & \texttt{w}::\texttt{g}, & \{\texttt{w},\texttt{y}\} & > \\
< & s_3, & \texttt{main}, & \texttt{x}::\texttt{g}, & \{\texttt{x},\texttt{z},\texttt{u}\} & > \\
< & s_4, & \texttt{y}::\texttt{y}, & \texttt{w}::\texttt{w}, & \perp & > \\
< & s_5, & \texttt{x}::\texttt{x}, & \texttt{w}::\texttt{w}, & \perp & > \\
< & s_6, & \texttt{z}::\texttt{z}, & \texttt{x}::\texttt{x}, & \perp & > \\
< & s_7, & \texttt{u}::\texttt{u}, & \texttt{z}::\texttt{z}, & \perp & > \\
< & s_8, & \texttt{u}::\texttt{u}, & \texttt{y}::\texttt{y}, & \perp & >
\end{array}
$$

Figure 3.11: The call instances for the PVG in Figure 3.10.

the set of call sites is

$$S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$$

where the s_i are arbitrary labels assigned to each call site. The set of root nodes is

$$R = \{\texttt{main}\}.$$

The function node `main` contains three call site sub-nodes: one for each constructor call invoked by the **new** operations, and one for the call to `g()` (the calls to the actual storage allocator have been elided for conciseness of presentation).

Note that `main` has three call *sites*, or subnodes, but four call *instances*, or edges. The call to `g()` could be to either `w`'s `g()` or `x`'s `g()`, since a variable of type pointer to `w` can actually point to `w` or any of its derived classes. Subsequent analysis can eliminate the call to `x::g()` as a possibility, either (1) by discovering that no `x, z`, or `u` objects are created, or (2) by discovering that the only possible type for `wp` at this program point is `w`. Our analysis will take the first approach, which we will show to be both simpler and faster, although not always as accurate.

A call instance edge is a tuple $< s, f, g, P >\in I$ where

- $s \in S$ is a call site,
- $f \in F$ is the *calling function*,
- $g \in F$ is the *target function* of the call, and
- If s is not a virtual call, $P = \perp$. If s is a virtual call site, then P is the set of *possible classes* upon which a virtual method call can be invoked ($P \subseteq C$, the set of all classes).

Figure 3.11 shows the call instances for our example program. For direct function calls, like the calls to the constructors for `w` and `y`, there is a one-to-one mapping of call sites to call instances. However, for indirect calls (virtual calls, function pointer calls, and pointer-to-member function calls), there may be many call instances per call site.

The virtual call to `g()` is represented by the two call instances for the call site s_3. The call instance for `w::g()` will be invoked when the dynamic type of the object pointed to by `wp` is `w` or `y`, so these types are in the first instance's set of possible classes. The call instance for `x::g()` will be invoked when the dynamic type of `ap` is `x, z`, or `u`.

If a call is made via a function pointer, there will be one call instance for each function in the program whose type matches the function pointer type. If a call is made via a pointer to a member function, there will be one call instance for each function in the class of the static object type, and its transitive base classes, that matches the function pointer type.

Since this dissertation is concerned primarily with virtual function calls, and since methods for handling calls via function pointers are well known (e.g. [Burke et al. 1994]), we will touch only briefly on function pointers and concentrate on virtual calls. Our implementation handles all call site types, and our benchmarks make use of all of them.

3.4.2 Building the PVG

Building the PVG requires the CHG, as well as information about the call sites in the source program. The input to the PVG builder algorithm is the CHG, the set of functions F, and the source-level call site information. In this and all other subsequent algorithms, the CHG and the associated *Inherit* and *Override* sets are considered to be "global variables" and are not explicitly passed as input parameters.

In practice, the source-level call site information is obtained by scanning the program during compilation, and is not actually placed in sets. However, to make this process abstract and to clarify exactly what source-level information is required, we will represent the source-level call site information in the set Σ.

The source-level call site information in Σ is contained in subsets representing the different types of call sites in the programs: *direct* call sites (Σ_D) and *virtual* call sites (Σ_V). Function pointer and pointer-to-member calls will be dealt with in the following section. All call sites are tuples in which

buildPVG(Σ_D, Σ_V)
    $I \leftarrow \emptyset$
    for each $< s, f, g > \in \Sigma_D$
        $I \leftarrow I \cup \{< s, f, g, \perp >\}$
    for each $< s, f, v > \in \Sigma_V$
        addVirtualInstances(s, f, v)

addVirtualInstances$(s \in S_V, f \in F, v \in V)$
    Let $< c, m, d >= v$
    if $< s, f, m, \mathit{Inherits}(v) > \in I$
        return
    $I \leftarrow I \cup \{< s, f, m, \mathit{Inherits}(v) >\}$
    for each $w \in \mathit{Override}(v)$
        addVirtualInstances(s, f, w)

Figure 3.12: Algorithm to Construct the Program Virtual-call Graph (PVG)

- $s \in S$ is a unique call site identifier, and
- $f \in F$ is the *calling function* (the source of the call)

A direct call site $k \in \Sigma_D$ is a tuple $< s, f, g >$ where

- $g \in F$ is the *target function* of the call.

A virtual call site $k \in \Sigma_V$ is a tuple $< s, f, v >$ where

- $v \in V$ is the *visible function* corresponding to the static type of the call.

The sets S_D and S_V are subsets of S which contain the call site identifiers of the direct and virtual call sites, respectively. Formally,

$$S_D = \{s \in S \mid f \in F, t \in F : < s, f, t > \in \Sigma_D\}$$

and

$$S_V = \{s \in S \mid f \in F, v \in V : < s, f, v > \in \Sigma_V\}.$$

The algorithm in Figure 3.12 constructs the PVG. The nodes of the PVG, the methods and functions of the set F, have been given. The algorithm adds edges to the set I of *call instances*. In essence, building the PVG is a process of converting source-level information about each call site into a set of possible targets for that call.

Lines 3 and 4 simply add the direct call edges to the PVG. Lines 5 and 6 invoke **addVirtualInstances** for each virtual call site. This recursive function adds a call instance corresponding to the statically declared type of the object through which the call is made. The set of possible classes is the *Inherits* set of the corresponding visible method node.

The function is then invoked recursively for each overriding function. The test at the beginning ensures that we do not needlessly re-traverse the CHG in cases of multiple inheritance.

3.4.3 Complexity

The expected cost of the **buildPVG** algorithm is

$$O(S_D + S_V D \log C)$$

That is, for each call site in S_V there could be as many as D (the number of edges in the CHG) recursive calls, each of which will cost $log\, C$ to look up and insert the call instance (we assume that in the implementation the call instances are attached to their associated call site).

In practice, the number of overrides will usually be small and the cost of set lookup will be constant.

3.4.4 Function Pointers

The PVG construction algorithm of the previous section only dealt with direct and virtual calls. While this is sufficient for some languages, like Java, C++ also allows calls through function pointers and pointers to member functions.

We define $\mathit{TypeOf}(f)$, where $f \in F$, to be the *type* of a function, according to the semantics of the language. The *signature* of a function, which we discussed previously, is a pair consisting of the function name and the function type. We let

$$T = \{\, \mathit{TypeOf}(f) \mid f \in F\}$$

be the set of all function types in the program.

Formally, as before, all source level call sites are tuples in which

- $s \in S$ is a call site, and
- $f \in F$ is the *calling function*

buildExtendedPVG$(\Sigma_D, \Sigma_V, \Sigma_P, \Sigma_M)$
    buildPVG(Σ_D, Σ_V)
    for each $< s, f, t > \in \Sigma_P$
        for each $g \in F :\ \mathit{TypeOf}(g) = t$
            $I \leftarrow I \cup \{< s, f, g, \perp >\}$
    for each $< s, f, c, b, t > \in \Sigma_M$
        addMemberInstances(s, f, b, t)
        for each $v \in V_c$
            Let $< c, m, d > = v$
            if $\mathit{IsVirtual}(m)$ **and** $\mathit{TypeOf}(m) = t$
                if $\exists_{n \in M, e \in C} :\ (< b, n, e > \in V$ **and** $\mathit{Sig}(m) = \mathit{Sig}(n))$
                    addVirtualInstances(s, f, v)

addMemberInstances$(s \in S_M, f \in F, c \in C, t \in T)$
    for each $m \in M :\ < c, m, c > \in V$ **and** $\mathit{TypeOf}(m) = t$
        if not $\mathit{IsVirtual}(m)$
            $I \leftarrow I \cup \{< s, f, m, \perp >\}$
    for each $b \in C :\ < b, c > \in D$
        addMemberInstances(s, f, b, t)

Figure 3.13: Building the PVG for C++: handling function pointer and pointer-to-member calls.

A function pointer call site $k \in \Sigma_P$ is a tuple $< s, f, t >$ where

- $t \in T$ is the *type* of the function pointer.

A pointer-to-member function call site $k \in \Sigma_M$ is a tuple $< s, f, c, b, t >$ where

- $c \in C$ is the *class* of the object or expression through which the call was made,
- $b \in \mathit{Bases}^*(c)$ is the class of the member function pointer, and
- $t \in T$ is the *type* of the function pointer.

The sets S_P and S_M are defined analogously to S_D and S_V to be the sets of call site identifiers of function pointer call sites and pointer-to-member function call sites, respectively.

Figure 3.13 extends the PVG construction algorithm to handle both types of calls. The call instances for a function pointer call are straightforward: they consist of one instance for each function whose type is equivalent to that used at the call site.

Member function pointers point to a function of the class that matches a particular type signature. For a given function type and a given class, the possible non-virtual member functions are all non-virtual functions of the class or its *base classes* that match the type; the possible virtual member functions are the virtual functions that are visible in the type of the member pointer. For further information see the C++ Annotated Reference Manual [Ellis and Stroustroup 1990].

The call instances for a pointer-to-member function call are calculated by starting at the static class type of the pointer and moving up the CHG. This is not a particularly efficient procedure for calculating the call instances, but pointer-to-member functions are rarely used and are unlikely to merit a more sophisticated implementation.

Complexity

The theoretical complexity of adding the function pointer call instances is $O(S_P F)$, but in practice S_P is likely to be small and the number of functions that match the type of any one of them is also likely to be small.

The theoretical complexity of adding the pointer-to-member call instances to the PVG is $O(S_M D \log \mathcal{M})$, but once again the number of such call sites is likely to be very small (we encountered only one benchmark which contained any such calls; there were 2). In addition, there are not likely to be a large number of base classes.

3.4.5 Constructing the PVG for Java

There are no special considerations for constructing the PVG for Java. We can simply use the algorithm of Figure 3.12. However, it is worth pointing out that Java programs may contain significant numbers of direct calls, either as static member functions or as methods that are introduced as `final`.

Chapter 4

The Rapid Type Analysis Algorithm

4.1 The Problem

The key to the optimization of a number of object-oriented language features lies in knowing to what types an object reference in the static text of the program might actually refer during the dynamic execution of the program.

More formally, for a particular expression e in the static program text, we would like to know the set $E \subseteq C$ of possible types for the object to which e will evaluate (or refer) at run-time.

This problem is related to the problem of type-inferencing in untyped languages. However, as discussed in Chapter 2, that term often refers to attempts to determine the type for the purpose of performing compile-time checks of the sort that statically typed languages typically perform. We therefore use the term *type analysis* to draw the distinction.

4.2 The Analysis Spectrum

A variety of algorithms have been proposed either to solve the type analysis problem, or to solve broader problems whose solution includes type analysis. There is considerable variation in complexity among these algorithms, both in the formal sense of computational complexity and in the pragmatic sense of complexity of implementation. Both types of complexity are an issue in compiler implementation: a slow optimization is less likely to

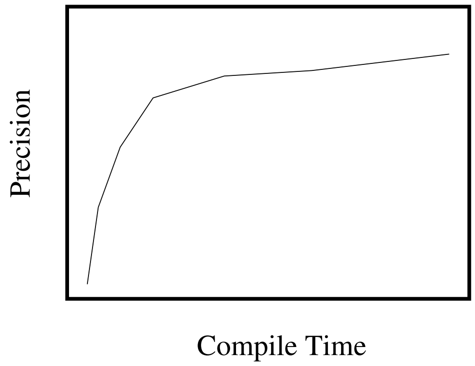


Figure 4.1: Type Analysis Algorithms: time to execute during compilation versus accuracy of solution. We are searching for a type analysis algorithm at the inflection point of this curve.

be used by programmers, and a more complicated algorithm is less likely to be added to a compiler, which is already a very complicated piece of software.

The optimizations described in Chapters 5, 6, and 7 all require the capability to ask for the set E to optimize a particular expression. As long as it can provide this information in a conservative manner, any type analysis algorithm can be used to drive the optimization.

Our hypothesis, which we will validate with our subsequent experimental results, is that *an algorithm exists for the type analysis problem which is close to optimal in precision, but is not very expensive to implement or execute.* This is shown conceptually in Figure 4.1: we are looking for the algorithm at the "inflection point" in the precision-time tradeoff graph.

The algorithm that we have developed to fill this role is called Rapid Type Analysis.

4.3 Static Analysis Overview

Before describing Rapid Type Analysis formally, in this section we will describe three fast static algorithms in the analysis spectrum. Two algorithms have been previously published in the literature (*Unique Name* [Calder and Grunwald 1994] and *Class Hierarchy Analysis* [Dean et al. 1995; Fernandez 1995]). The third algorithm, *Rapid Type Analysis*,

```
class A {
  public:
    virtual int foo() { return 1; };
};

class B: public A {
  public:
    virtual int foo() { return 2; };
    virtual int foo(int i) { return i+1; };
};

void main() {
    B* p = new B;
    int result1 = p->foo(1);
    int result2 = p->foo();
    A* q = p;
    int result3 = q->foo();
}
```

Figure 4.2: Program illustrating the difference between the static analysis methods.

is new. We will sometimes abbreviate the algorithms as UN, CHA, and RTA, respectively.

We use a small example program to illustrate the three algorithms and give some intuition about how RTA works. We then briefly compare them in power to other static analyses, and discuss the interaction of type safety and analysis.

4.3.1 Unique Name

The first published study of virtual function call resolution for C++ was by Calder and Grunwald [1994]. They were attempting to optimize C++ programs at link time, and therefore had to confine themselves to information available in the object files. They observed that in some cases there is only one implementation of a particular virtual function anywhere in the program. This can be detected by comparing the mangled names [1] of the C++ functions in the object files.

When a function has a unique name (really a unique signature), the virtual call is replaced with a direct call. While it can be used within a compiler in the same manner

[1]The *mangled name* of a function is the name used by the linker. It includes an encoding of the class and argument types to distinguish it from other identically named functions.

as the other algorithms evaluated in this chapter, Unique Name has the advantage that it does not require access to source code and can optimize virtual calls in library code. However, when used at link-time, Unique Name operates on object code, which inhibits further optimizations such as inlining.

Figure 4.2 shows a small program which illustrates the power of the various static analyses. There are three virtual calls in `main()`. Unique Name is able to resolve the first call (that produces `result1`) because there is only one virtual function called `foo` that takes an integer parameter – `B::foo(int)`. There are many `foo` functions that take no parameters, so it can not resolve the other calls.

4.3.2 Class Hierarchy Analysis

Class Hierarchy Analysis [Dean et al. 1995; Fernandez 1995] uses the combination of the statically declared type of an object with the class hierarchy of the program to determine the set of possible targets of a virtual function call. In Figure 4.2, `p` is a pointer whose static type is `B*`. This means that `p` can point to objects whose type is `B` or any of `B`'s derived classes.

By combining this static information with the class hierarchy, we can determine that there are no derived classes of `B`, so that the only possible target of the second call (that produces `result2`) is `int B::foo()`.

Class Hierarchy Analysis is more powerful than Unique Name for two reasons: it uses class type declarations (as in Figure 4.2), and it can ignore identically-named functions in unrelated classes.

Class Hierarchy Analysis must have the complete program available for analysis, because if another module defines a class `C` derived from `B` that overrides `foo()`, then the call can not be resolved.

In the process of performing Class Hierarchy Analysis, we build a call graph for the program. The call graph includes functions reachable from `main()` as well as those reachable from the constructors of global-scope objects. Note that some other researchers use the term "Class Hierarchy Analysis" to denote only the resolution of virtual calls, not the building of the call graph.

4.3.3 Rapid Type Analysis

Rapid Type Analysis starts with a call graph generated by performing Class Hierarchy Analysis. It uses information about instantiated classes to further reduce the set of executable virtual functions, thereby reducing the size of the call graph.

For instance, in Figure 4.2, the virtual call `q->foo()` (which produces `result3`) is not resolved by Class Hierarchy Analysis because the static type of `q` is `A*`, so the dynamic type of the object could be either `A` or `B`. However, an examination of the entire program shows that no objects of type `A` are created, so `A::foo()` can be eliminated as a possible target of the call. This leaves only `B::foo()`.

Note that RTA must not count the invocation of a base class constructor as an actual instantiation of the base class: when an object of type `B` is created, `A`'s constructor is called to initialize the `A` sub-object of `B`. However, the virtual function table of the contained object still points to `B`'s `foo()` method.

Rapid Type Analysis builds the set of possible instantiated types *optimistically*: it initially assumes that no functions except `main` are called and that no objects are instantiated, and therefore no virtual call sites call any of their target functions. It traverses the call graph created by Class Hierarchy Analysis starting at `main`. Virtual call sites are initially ignored. When a constructor for an object is found to be callable, any of the virtual methods of the corresponding class that were left out are then traversed as well. The live portion of the call graph and the set of instantiated classes grow iteratively in an interdependent manner as the algorithm proceeds.

Rapid Type Analysis inherits the limitations and benefits of Class Hierarchy Analysis: it must analyze the complete program. Like CHA, RTA is flow-insensitive and does not keep per-statement information, making it very fast.

Rapid Type Analysis is designed to be most effective when used in conjunction with class libraries. For instance, a drawing library defines numerous objects derived from class `shape`, each with their own `draw()` method. A program that uses the library and only ever creates (and draws) squares will never invoke any of the methods of objects like `circle` and `polygon`. This will allow calls to `draw()` to be resolved to calls to `square::draw()`, and none of the other methods need to be linked into the final program. This leads to both reduced execution time and reduced code size.

Another approach to customizing code that uses class libraries is to use class slicing [Tip et al. 1996].

4.3.4 Other Analyses

There are several other levels of static analysis that can be performed. First, a simple local flow-sensitive analysis would be able to resolve this call:

```
A* q = new B;
q = new A;
result = q->foo();
```

because it will know that `q` points to an object of type `A`. Rapid Type Analysis would not resolve the call because both `A` and `B` objects are created in this program.

An even more powerful static analysis method is alias analysis, which can resolve calls even when there is intervening code which could potentially change an object's type. Alias analysis is discussed more fully in Section 5.5.2, with related work.

4.3.5 Type Safety Issues

An important limitation of CHA and RTA is that they rely on the type-safety of the programs. Continuing to use the class hierarchy from Figure 4.2, consider the following code fragment:

```
void* x = (void*) new B;
B* q = (B*) x;
int case1 = q->foo();
```

Despite the fact that the pointer is cast to `void*` and then back to `B*`, the program is still type-safe because we can see by inspection that the down-cast is actually to the correct type. However, if the original type is `A`, as in

```
void* x = (void*) new A;
B* q = (B*) x;
int case2 = q->foo();
```

then the program is not type-safe, and the compiler would be justified in generating code that raises an exception at the point of the virtual function call to `foo()`. However, because `foo()` is in fact defined for `A`, most existing compilers will simply generate code that calls `A::foo()`; this may or may not be what the programmer intended. If the call had instead been

```
int case3 = q->foo(666);
```

then the program will result in a undefined run-time behavior (most likely a segmentation fault) because `A`'s virtual function table (VFT) does not contain an entry for `foo(int)`.

The computation of `case1` is clearly legal, and the computation of `case3` is clearly illegal. In general it is not possible to distinguish the three cases statically. Unfortunately, in `case2`, Class Hierarchy Analysis would determine that the call was resolvable to `B::foo()`, which is incorrect. Rapid Type Analysis would determine that there are no possible call targets, which is correct according to the C++ language definition but different from what is done by most compilers.

Therefore, Class Hierarchy Analysis and Rapid Type Analysis either need to be disabled whenever a downcast is encountered anywhere in the program, or they can be allowed to proceed despite the downcast, with a warning printed to alert the programmer that optimization could change the results of the program if the downcasts are truly unsafe (as in `case2` or `case3`).

We favor the latter alternative because downcasting is very common in C++ programs. Additional flexibility can be provided by pragmas or compiler switches which allow virtual function call resolution to be selectively disabled at a call site or for an entire module. We will discuss this issue further when we present the results for one of our benchmarks, `lcom`, which contained some unsafe code.

Of course, this entire issue is absent in type-safe languages, such as Java, in which down-casts are always type-checked.

4.4 The Algorithm

The Rapid Type Analysis (RTA) algorithm walks over the PVG, starting at the root, and finds the set of classes instantiated in live code. The set of instantiated classes is intersected with the set of possible classes (derived from analysis of the CHG) contained in each virtual call instance to determine whether that call instance is potentially live. RTA's type analysis is crude, in that no flow analysis is performed within a procedure, and no attempt is made to differentiate the possible types which may be assigned to different variables. However, as will be seen, RTA is very effective.

Figure 4.3 shows a recursive formulation of the algorithm. In addition to the PVG, it makes use of four sets, which represent the classes (C_L), functions (F_L), call sites (S_L), and call instances (I_L) that are *live* in the program. A class is live if it may be created by

```
rapidTypeAnalysis(F, S, I, R)
    Q_V ← ∅
    C_L ← F_L ← S_L ← I_L ← ∅
    for each f ∈ R
        analyze(f, false)

analyze(f ∈ F, isbase ∈ Boolean)
    if IsConstructor(f) and not isbase
        instantiate(ClassOf(f))
    if f ∈ F_L
        return
    F_L ← F_L ∪ {f}
    for each s ∈ S, t ∈ F, P ∈ 2^C : < s, f, t, P >∈ I
        Let i =< s, f, t, P >
        if s ∈ S_D or (s ∈ S_V and C_L ∩ P ≠ ∅)
            addCall(i)
        else
            addVirtualMappings(P, i)

addCall(i ∈ I)
    Let < s, f, t, P >= i
    I_L ← I_L ∪ {i}
    S_L ← S_L ∪ {s}
    analyze(t, IsBaseConstructorCall(i))

instantiate(c ∈ C)
    if c ∈ C_L
        return
    C_L ← C_L ∪ {c}
    for each i ∈ I : < c, i >∈ Q_V
        if i ∉ I_L
            addCall(i)
            Q_V ← Q_V − {< c, i >}

addVirtualMappings(P ∈ 2^C, i ∈ I)
    for each p ∈ P
        Q_V ← Q_V ∪ {< p, i >}
```

Figure 4.3: The Rapid Type Analysis Algorithm

the program; a function is live if it can be reached from the root of the PVG; a call site is live if any of its call instances are live; and a call instance is live if its enclosing function is live and, if it is a virtual call, if one of its possible classes ($p \in P$) is live.

The algorithm also maintains a mapping from classes to call instances. This is because we may encounter a virtual call that can be invoked on classes `a` and `b`, which we have not yet discovered to be live, but will later discover to be live. In such a situation, we add an entry for each of the two classes to the mapping, which is checked later when new class instantiations are discovered.

After the program has been analyzed, a post-pass over the call sites constructs S_R, the set of call sites that have only one live target, and have therefore been resolved into direct calls.

The **analyze** function of the RTA algorithm analyzes a function body. It begins by checking whether the function is a constructor, and that constructor is not being invoked by the constructor of a derived class (line 7). If that is so, it calls **instantiate** on the class of the constructor function f, which takes care of adding the class to C_L, the set of live classes (line 8).

Then **analyze** checks whether f has already been processed, and if so returns without performing further work (lines 9 and 10). This check must be done *after* the instantiation check for constructors because it is possible that the first time the constructor is visited is when it is a base constructor call, but is subsequently called to construct its own class.

If this is the first time that f has been examined, f is then added to F_L, the set of live functions (line 11).

On line 12, **analyze** iterates over the call instances of f. Direct calls are always added to the live call graph; virtual calls are only added if one of the possible classes in P_x for the method is in the set of live classes C_L (lines 14 and 15). If none of the possible classes is live, then one entry for each of the possible classes is made in the map Q_V (line 17). This way, if one of the classes in P_x is subsequently added to C_L, the call instance will be added to the live call graph because the class will be found in the map Q_V.

The **addCall** function takes a call instance and adds it to the set of live instances I_L, and adds the call site of the call instance to the set of live call sites S_L. It then calls **analyze** on the target t of the call instance (line 22).

The second argument to **analyze** is a boolean flag that tells whether the call is a call from a constructor of a derived class to a constructor of its base class. In this case, we do not want to treat this call as one that causes objects of the case class to become instantiated.

The **instantiate** function does the work of adding a new live class. If the class has already been added to C_L, it simply returns without performing any work (lines 24 and 25). Otherwise, the class c is added to the set of live classes C_L (line 26).

Now if there are any entries in the map Q_V for call instances that were not added to the live call graph because class c had not yet been instantiated, those instances are added to the live call graph (lines 27–30). A check to make sure that the instance has not already been added is necessary because some other class in the set P_x could have been instantiated previously.

4.5 Complexity

The complexity of the RTA algorithm is relatively simple to analyze if we use a few counting tricks. The **analyze** function is called for every root function and for every live call instance, so it is called at most $R + I_L$ times. We assume that the set of roots of the call graph is a small constant, or that there is a single artificial root node with direct call instances to all of the actual root calls; the former situation exists for Java, and the latter is the approach we took in our implementation for C++.

By similar logic, **addCall** is only called once for each instance in I_L. And since the set union operations it contains are actually implemented by simply turning on bits in the call site and call instance objects, the total cost of all calls to **addCall** (excluding the functions it calls) is $O(I_L)$.

The **instantiate** function is called from **analyze**, and could be called once for each live constructor call in the program. The number of live constructor calls is certainly less than the total number of live calls, I_L.

The **analyze** function is called I_L times, and the membership and union operations on the set F_L take constant time since F_L is implemented with with a flag on each member of F. The loop on line 12 iterates over all of the call instances originating from the current function f. However, since this part of **analyze** is executed at most once per function, we can bound the number of total iterations of the loop for *all* executions of **analyze** by I, the total number of call instances.

Now let us examine the loop on lines 13–17. The **if** test take constant time if $s \in S_D$, and $O(P)$ time if $s \in S_V$ (since C_L is implemented as a flag on each member of C, we perform P constant-time checks). The **else** branch of the statement inserts P items into Q_V, and each insertion takes $O(C)$ worst-case time because Q_V is represented by a balanced tree and all entries for the same class p are linked to a single tree node for that class.

If we define $\mathcal{P}$ to be the maximum size of all sets P, or

$$\mathcal{P} = \max_{<s,f,g,P>\in I} |P|$$

then the time for all iterations of the loop is $O(I\mathcal{P}\log C)$. Therefore, the total cost for all executions of **analyze** is $O(R + I_L + I\mathcal{P}\log C)$.

Now we only have to account for the executions of the body of **instantiate**, which is called $O(I_L)$ times. The test if $c \in C_L$ takes constant time because membership in C_L is implemented with a flag for each member of C. Therefore, the body of **instantiate** from lines 26–30 is only executed C_L times.

The union and membership operations on lines 26 and 28 take constant time due to the use of flags. Recall that all call instances in the map Q_V associated with class c are kept in a list attached to the node for c in Q_V. Therefore, it takes $O(\log C)$ time to find the node (in line 27) and to remove it (on line 30 – which is really a single operation, outside of the loop).

For all executions of **instantiate**, the body of the loop is executed at most once for each element of Q_V, and we determined in our analysis of the **analyze** function that $O(I\mathcal{P})$ entries are added to Q_V. Therefore, the total time for all executions of **instantiate** is $O(I_L + C_L \log C + I\mathcal{P})$.

We can now determine the total running time for the RTA algorithm by adding the costs for all three component functions. The result is

$$O((R + I_L + I\mathcal{P}\log C) + I_L + (I_L + C_L \log C + I\mathcal{P})$$

which can be simplified to

$$O(R + I\mathcal{P}\log C + C_L \log C)$$

and by observing that since there must be at least one call instance for the constructor call to each class in C_L then $|C_L| \leq |I|$, we arrive at a final worst-case complexity of

$$O(R + I\mathcal{P}\log C).$$

4.5.1 Expected Complexity

In all but the most unusual programs the number of root functions of the call graph is small, so the R term can be dropped.

By implementing the map structure Q_V with a hash table, we can reduce the expected complexity to $O(I\mathcal{P})$ at the cost of degrading the worst-case complexity to $O(I\mathcal{P}C)$.

The $I\mathcal{P}$ term is due to the fact that in the worst case, we must assume that every virtual instance is added to Q_V. There are a number of observations we can make about this term. First, direct calls are never added to Q_V, so $(I - S_D)\mathcal{P}$ is a more accurate bound on the maximum number of entries. Second, in most cases the sets P of the virtual call instances will be small, since it is rare for a large number of classes to inherit a virtual function. Finally, entries are added to Q_V only if none of the classes in P have been created. Unless program organization is unusual, classes will be created "before" they are used (in the sense of the call graph), and therefore their virtual calls will not be added to Q_V at all.

Because of all of these factors, we expect the time for RTA in practice to be proportional to the size of I.

4.6 Function and Member Function Pointers

The RTA algorithm presented in the previous section only handles programs with direct and virtual calls. We now extend the RTA algorithm to handle programs with function pointer and pointer-to-member calls.

The extension for function pointers is straightforward, using techniques developed previously to build call graphs for C and Fortran programs. A function can be called by pointer only if its address is taken. During our initial scan of the program (when we identify the call sites), we also make note of any functions whose addresses are taken in a function. This set is denoted $\mathit{FunctionPointers}(f)$, where $f \in F$. Even in programs that make use of function pointer calls, $\mathit{FunctionPointers}(f) = \emptyset$ for most functions f.

The extended RTA algorithm is shown in Figures 4.4 and 4.5. When a function f is analyzed, $\mathit{FunctionPointers}(f)$ is added to the set F_A, which is the set of functions whose addresses have been taken in live code.

When a function pointer call is encountered, there will be a set of call instances corresponding to the functions in F whose type matches the type of the function pointer.

Those call instances whose corresponding functions are in F_A are added to the call graph; those that are not are added to Q_P, which keeps track of function pointer call instances that may subsequently have to be added to the live call graph. When a new function g is added to F_A, any entries in Q_P for g are added to the live call graph.

Member function pointers are handled in an identical manner. However, since a regular function pointer can not be used for a pointer-to-member function call (and vice versa), member function pointers must be handled separately.

4.7 Special Issues for C++

In our presentation of the RTA algorithm, we have omitted a few details of the C++ language that complicate analysis.

4.7.1 Construction VFT's

Construction virtual function tables are an arcane feature of C++ designed to allow virtual functions to be called from constructors of partially constructed objects. Unfortunately, this feature complicates analysis and sometimes reduces its precision.

Consider the code in Figure 4.6: when an object of type `B` is created, `A`'s constructor will be invoked. `A`'s constructor calls the virtual function `foo()`. Since the `B` portion of the object is uninitialized while `A`'s constructor executes, it would be erroneous to invoke `B`'s `foo()` method.

Therefore, C++ mandates that in such circumstances, a special virtual function table be created for the time during which the `B` object is partially constructed. This table is known as the construction VFT.

Construction VFT's create a problem for type analysis algorithms for C++. In the scenario just described, even though no `A` objects are created, an object is alive in the program which acts like an `A` object for purposes of virtual function dispatch. This means that as presented, the algorithm might not build an accurate call graph.

The solution that we adopt is relatively straightforward. It is not the most precise solution, but it is in the spirit of the RTA algorithm in that it is a fast solution that will work in the most common cases.

When the code of each method is first analyzed to identify the function call sites, we now also collect an additional piece of information: if the `this` pointer is copied or passed

```
extendedRTA(F, S, I, R)
    Q_V ← Q_P ← Q_M ← ∅
    C_L ← F_L ← S_L ← I_L ← ∅
    F_A ← M_A ← ∅
    for each f ∈ R
        analyze(f, false)

analyze(f ∈ F, isbase ∈ Boolean)
    if IsConstructor(f) and not isbase
        instantiate(ClassOf(f))
    if f ∈ F_L
        return
    F_L ← F_L ∪ {f}
    addFunctionPointers(f)
    addMemberPointers(f)
    for each s ∈ S, t ∈ F, P ∈ 2^C : < s, f, t, P >∈ I
        Let i =< s, f, t, P >
        if s ∈ S_D or (s ∈ S_V and C_L ∩ P ≠ ∅)
            addCall(i)
        else if s ∈ S_V
            addVirtualMappings(P, i)
        else if s ∈ S_P
            if t ∈ F_A
                addCall(i)
            else
                Q_P ← Q_P ∪ {< t, i >}
        else if s ∈ S_M
            if t ∈ M_A and (P =⊥ or P ∩ C_L ≠ ∅)
                addCall(i)
            else if t ∉ M_A
                Q_M ← Q_M ∪ {< t, i >}
            else
                addVirtualMappings(P, i)
```

Figure 4.4: RTA Extensions for Function Pointers and Member Function Pointers (1)

addFunctionPointers$(f \in F)$
    for each $g \in FunctionPointers(f) : \ g \notin F_A$
        $F_A \leftarrow F_A \cup \{g\}$
        for each $i \in I : < g, i > \in Q_P$
            $Q_P \leftarrow Q_P - \{< g, i >\}$
            addCall(i)

addMemberPointers$(f \in F)$
    for each $m \in MemberPointers(f) : \ m \notin M_A$
        $M_A \leftarrow M_A \cup \{m\}$
        for each $i \in I : < m, i > \in Q_M$
            $Q_M \leftarrow Q_M - \{< m, i >\}$
            Let $< s, g, h, P >= i$
            if $P = \perp$ **or** $P \cap C_L \neq \emptyset$
                addCall(i)
            else
                addVirtualMappings(P, i)

Figure 4.5: RTA Extensions for Function Pointers and Member Function Pointers (2)

```
class A {
  public:
    A::A() { foo(); };
    virtual void foo();
};

class B: public A {
  public:
    virtual void foo();
};
```

Figure 4.6: C++ Code Requiring Construction VFT's

as a parameter (other than the implicit `this` pointer), we mark a flag associated with the method, *ThisEscapes*, as **true**. Otherwise, *ThisEscapes* is marked **false**.

After constructing the PVG, but before invoking the Rapid Type Analysis procedure, we make a bottom-up pass over the PVG. When there is a call instance from method m to method n via the `this` pointer of m, if n is not a base class constructor then

$$\mathit{ThisEscapes}(m) \leftarrow \mathit{ThisEscapes}(m) \ \textbf{or} \ \mathit{ThisEscapes}(n).$$

We now modify the RTA algorithm of Figure 4.3 so that line 22 reads

$$\textbf{analyze}(t, \text{IsBaseConstructorCall}(i) \ \textbf{and not} \ \mathit{ThisEscapes}(i))$$

Note that the mechanism for handling construction VFTs depends upon the fact that it is illegal to access an object that is partially constructed through any means except its `this` pointer. The C++ *Annotated Reference Manual* §12.1 states: "The type system makes it hard, but not impossible, to use an object before it is fully constructed. Use of a partially constructed object is undefined since it would violate any implicit or explicit invariants assumed about objects of the class" [Ellis and Stroustrup 1990].

4.7.2 Local Classes

The ability to define classes locally within a function body also complicates analysis, although purely in a practical rather than a theoretical fashion. Because RTA relies on knowing the entire class hierarchy to perform optimization, it means that it is not possible to merely scan the class definitions, build the CHG, and then scan, translate, and optimize each procedure in a single pass.

We have found nested classes to be relatively rare. Because we wish to be able to compile large programs efficiently, we optimistically assume that there are no local classes, build the CHG from the globally available class definitions, and then compile each function. If any local classes are found, then any functions that were optimized using the RTA information are re-compiled using a "fast path" which avoids performing many of the checks associated with normal compilation. In particular, static semantic checks can be omitted since the functions in question have already been compiled once and determined to conform to the language rules.

4.8 Adapting RTA for Java

Once the CHG and PVG have been constructed, most of the language-specific aspects have been dealt with. There are only a few minor issues that are specific to Java.

Mostly, Java is simpler than C++. There are no construction VFTs, so the *ThisEscapes* calculation and related machinery described in the previous section are not needed.

One difference between C++ and Java is that Java has a common ancestor for all implementation classes (the class `Object`). The sets of possible methods P_x for call sites through the `Object` type will be very large, including most or all of the classes in C.

As a result, the **analyze** function could potentially spend a great deal of time uselessly adding entries to the map Q_V (lines 14 and 17 of Figure 4.3). The solution to this problem is simple: in the initialization on line 3, the set C_L is initialized to $\{\texttt{Object}\}$ instead of $\emptyset$. As a result, when a call is made through a pointer of type `Object`, the condition $C_L \cap P_x \neq \emptyset$ on line 14 will always be true and line 17 will not be executed.

Treating `Object` as an instantiated class will not degrade the performance of the algorithm, since even if some classes override methods of the class `Object`, there will always be some class in C_L that *does* inherit the default methods.

It is possible that as a result of these differences, Java would be more likely than C++ to derive improvement in precision from a flow-sensitive algorithm for type analysis.

Chapter 5

Resolution of Virtual Function Calls

A major advantage of object-oriented languages is abstraction. The most important language feature that supports abstraction is the dynamic dispatch of methods based on the run-time type of an object. In dynamically typed languages like Smalltalk and SELF, all dispatches are considered dynamic, and eliminating these dynamic dispatches has been essential to obtaining high performance [Chambers et al. 1991b; Hölzle et al. 1991; Ungar et al. 1992].

C++ is a more conservatively designed language. Programmers must explicitly request dynamic dispatch by declaring a method to be virtual. C++ programs therefore suffer less of an initial performance penalty, at the cost of reduced flexibility and increased programmer effort. However, virtual function calls still present a significant source of opportunities for program optimization.

The most obvious opportunity, and the one on which the most attention has been focused, is execution time overhead. Even with programmers specifying virtual functions explicitly, the execution time overhead of virtual function calls in C++ has been measured to be as high as 40% [Lee and Serrano 1995]. In addition, as programmers become familiar with the advantages of truly object-oriented design, use of virtual functions increases. The costs associated with developing software are so high that the performance penalty of virtual functions is often not sufficient to deter their use. Therefore, unless compilers are improved, the overhead due to virtual function calls is likely to increase as programmers make more extensive use of this feature.

Other researchers have shown that virtual function call resolution can result in significant performance improvements in execution time performance for C++ programs [Calder and Grunwald 1994; Aigner and Hölzle 1996; Lee and Serrano 1995]; in our work we concentrate on comparing algorithms for resolving virtual function calls, and investigating the reasons for their success or failure.

Another opportunity associated with virtual functions is code size reduction. For a program without virtual function calls (or function pointers), a complete call graph can be constructed and only the functions that are used need to be linked into the final program. With virtual functions, each virtual call site has multiple potential targets. Without further knowledge, all of those targets and any functions they call transitively must be included in the call graph.

As a result, object-code sizes for C++ programs have become a major problem in some environments, particularly when a small program is statically linked to a large object library. For instance, when a graphical "hello world" program is statically linked to a GUI object library, even though only a very small number of classes are actually instantiated by the program, the entire library can be dragged in.

Finally, virtual function calls present an analogous problem for browsers and other program-understanding tools: if every potential target of a virtual function call is included in the call graph, the user is presented with a vastly larger space of object types and functions that must be comprehended to understand the meaning of the program as a whole.

In this chapter, we compare three fast static analysis algorithms for resolving virtual function calls and evaluate their ability to solve the problems caused by virtual function calls in C++. We also use dynamic measurements to place an upper bound on the potential of static analysis methods, and compare the analysis algorithms against more sophisticated analyses like alias analysis. Finally, we present measurements of the speed of the analysis algorithms, which demonstrate that they are fast enough to be included in commercial-quality compilers.

5.1 Algorithms for Resolution of Indirect Calls

Once an algorithm like Rapid Type Analysis has been applied to a program, resolving virtual calls into direct calls is relatively straightforward. Since Rapid Type Analysis

resolveCalls(F, S_L, I_L)
    $S_R \leftarrow \emptyset$
    for each $s \in S_L : s \notin S_D$
        $Q \leftarrow \{< s, f, t, P >\in I_L \mid f \in F, t \in F, P \in 2^C\}$
        if $|Q| = 1$
            $S_R \leftarrow S_R \cup \{s\}$

(a) Indirect call resolution algorithm for use with RTA.

resolveVirtualCalls(F, S_V, I)
    $S_R \leftarrow \emptyset$
    for each $s \in S_V$
        $Q \leftarrow \{< s, f, t, P >\in I \mid f \in F, t \in F, P \in 2^C : P \cap E_s \neq \emptyset\}$
        if $|Q| = 1$
            $S_R \leftarrow S_R \cup \{s\}$

(b) Virtual function call resolution algorithm for use with any analysis.

Figure 5.1: Two algorithms for resolving indirect function calls. Algorithm (a) will only work with RTA, but it resolves all kinds of indirect calls (virtual, function pointer, and pointer-to-member). Algorithm (b) is completely general and only requires that analysis somehow calculate a set $E_s \subseteq C$ of possible classes for each virtual call site s.

computes a live call graph, it is implicitly determining the set of callable functions at each virtual call site. Therefore, resolving the virtual calls is simply a matter of going back over each virtual call site and determining whether there is only one possible function call target.

Figure 5.1 shows two algorithms for resolving virtual calls in the context of the notation established in the previous chapters. If RTA has been used, then the second algorithm is more straightforward: it simply says that any call site that has only one potential target (live call instance) can be resolved. This algorithm works for all types of calls, not just virtual calls, and is implemented as a simple post-pass to the RTA algorithm.

However, if some other algorithm has been used to compute the set of live classes, then the algorithm of Figure 5.1(b) can be employed. Class Hierarchy Analysis (which will be described in more detail later in this chapter) is a degenerate case in which we simply assume $C_L = C$.

5.2 Resolving Virtual Calls

So far we have assumed that resolving a call is a simple matter, and that all that we need to do is to identify the proper statically bindable target function and the problem is solved. Unfortunately, performing the actual resolution is not always straightforward.

Consider the program example in Figure 4.2 on page 42. The first two calls can be resolved as

```
B* p = new B;
int result1 = p->B::foo(1);
int result2 = p->B::foo();
```

because the static type of `p` is `B`, the same type to which the virtual call is being resolved. The third call can also be resolved by RTA to a call to `B::foo()` because there are no `A` objects created by the program. However, the resolution can not simply be

```
A* q = p;
int result3 = q->B::foo();
```

because the type of `q` is `A`, so the resolved expression would contain a type error. When the call is resolved to a method of a derived class of the static class type of the call expression, a down-cast must be included in the modified call expression. The correct resolution is

```
A* q = p;
int result3 = ((B*) q)->B::foo();
```

There are two potential problems that can result from the down-cast. First of all, in the example above, the class `B` might not be in scope at the point of the call because its definition is in a separate include file. If the optimization is being done in the back-end, this is not a problem provided that the address of `B::foo()` is available.

If the optimization is being performed as a source-to-source transformation, then the absence of `B`'s definiton will prevent the application of the optimization unless the compiler has access to *all* class definitions, not just those that are in scope. The compiler in which we implemented RTA, Montana, is an example of this type: it keeps all global class declarations in a "code store" [Barton et al. 1994] that is available at all phases of the compilation process. Note that even in such an environment, if the class `B` were a *local class* of some function, then the optimization could not be applied.

The second problem raised by the down-cast is that if there were multiple possible dynamic types for `q`, the adjustment to the `this` pointer might be variable even though the target function had been resolved. This problem will only occur when casting from a virtual base class to a derived class, which is disallowed in C++ for the static cast operator. (Note that the problem does not occur in the standard Sun object model for Java, because there are never any adjustments to the `this` pointer).

Section 7.4 describes how the type information computed by RTA can be used to determine if the down-cast can be applied statically. If this is not possible, the optimization can still be performed if the standard two-column object model of C++ [Ellis and Stroustroup 1990] is used: the `this` pointer adjustment can be calculated dynamically from the offsets in the virtual function table, just as though a virtual call were being performed, and then the statically bound call can be made. However, with a thunk object model (in which a non-zero `this` pointer adjustment is performed by a thunk which then jumps to the target function), the necessary information is unavailable and the optimization can not be performed.

In practice, the down-casting problem is quite rare. However, if the compiler does not have a "global" view of the class hierarchy, the scoping problem may significantly limit optimization potential.

5.3 Software Architecture

This section describes the architecture of the software for performing both the optimizations and the measurements described in this chapter.

5.3.1 Optimizer Architecture

To evaluate our optimizations, both static analysis of the benchmark's source code and dynamic analysis of its execution are required. Our system's architecture is illustrated in Figure 5.2.

A source program is fed into the prototype C++ compiler, which parses and type-analyzes the program. The intermediate form generated by the compiler is read in by our static analysis system, which generates the CHG and PVG, and then applies the Rapid Type Analysis algorithm. The output from this is a list of virtual call sites that can be resolved and a list of classes that need not have a virtual function table (VFT).

Since the prototype compiler does not yet generate code, this information is passed to the Eliminator, which modifies the source code to resolve the virtual calls. This code is

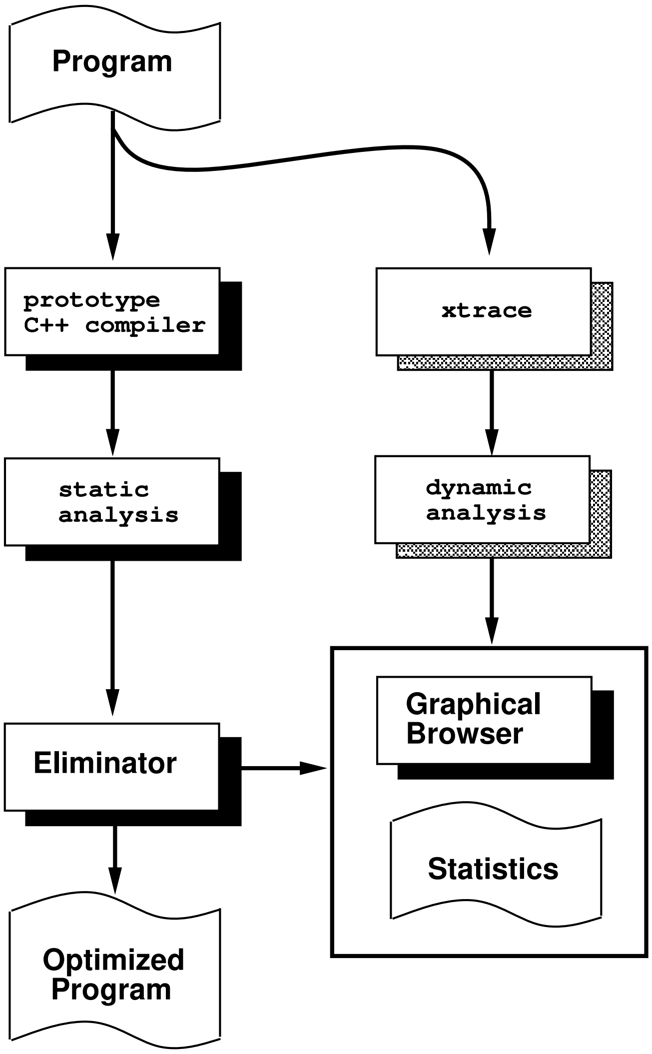


Figure 5.2: Eliminator architecture

then compiled and executed to verify that the resolved virtual functions do not change the results of the program, and to obtain speedup information.

The Eliminator was only able to generate compilable code for a few of the benchmarks because when the calls are resolved at the source level, many of the resolved calls are to methods of subclasses that are not in scope at the point of call. Therefor, the output of the Eliminator was used primarily to test the correctness of our transformation, rather than to provide comprehensive speedup numbers.

5.3.2 Measurements Architecture

Along with the optimizer, a number of other components allow us to collect static and dynamic statistics about the benchmarks. The source program is compiled with the standard xlC compiler and traced with the xtrace tool [Nair 1994], which is similar in design to QPT. This allows us to collect statistics on run-time calling behavior, which is combined with the results of the static analysis to determine how many virtual calls are being eliminated at run-time by our analysis. Information from the executable file image is also collected for use in code size reduction measurements.

One difficulty with collecting these measurements is that inlining, which is pervasive in C++ programs, changes the structure of the call graph. So to get frequency counts for the virtual call sites, we had to compile the programs with the "`-g`" option, which turns off optimization and includes line number tables in the executable file.

Since virtual call sites are not resolved by the existing compiler, the number of virtual calls is independent of whether the program is compiled with optimization. We therefore used the optimized executable to obtain statistics about non-virtual calls, and the unoptimized executable to obtain statistics about virtual calls. In the process, we discovered that in one application, 65% of the function calls were being inlined!

5.3.3 Timings

All timing trials were conducted on an unloaded IBM RS/6000 model 41T workstation, which contains an 80 MHz PowerPC 601 processor, 64 MB of RAM, and 2 GB of disk. Trials were run twelve times, the first and last result discarded, and the remaining trials averaged.

The operating system was AIX version 3.2.5; the compiler was xlC (CSet++) version 3.1.32.

5.4 Experimental Results

In this section we evaluate the ability of the three fast static analysis methods to solve the problems that were outlined in the introduction: execution time performance, code size, and perceived program complexity. Where possible, we will use dynamic measurement information to place an upper limit on what could be achieved by perfect static analysis.

5.4.1 Methodology

Our measurements were gathered by reading the C++ source code of our benchmarks into a prototype C++ compiler being developed at IBM. After type analysis is complete, we build a call graph and analyze the code. Since the prototype compiler is not yet generating code reliably enough to run large benchmarks, we compile the programs with the existing IBM C++ compiler on the RS/6000, xlC. The benchmarks are traced, and their executions are simulated from the instruction trace to gather relevant execution-time statistics. We then use line number and type information to match up the call sites in the source and object code.

We used both optimized and unoptimized compiled versions of the benchmarks. The unoptimized versions were necessary to match the call sites in the source code and the object code, because optimization includes inlining, which distorts the call graph. However, existing compilers can not resolve virtual function calls, so optimization does not change the number of virtual calls, although it may change their location, especially when inlining is performed. Therefore, turning optimization (and inlining) off does not affect our measurements of the number of resolved virtual function calls. Unoptimized code was only used for matching virtual call sites. All measurements are for optimized code unless otherwise noted.

Because our tool analyzes source code, virtual calls in library code were not available for analysis. Only one benchmark, `simulate`, contained virtual calls in the library code. They are not counted when we evaluate the efficacy of static analysis, since had they been available for analysis they might or might not have been resolved.

The information required by static analysis is not large, and could be included in compiled object files and libraries. This would allow virtual function calls in library code to be resolved, although it would not confer the additional benefits of inlining at the virtual call site.

Benchmark	Lines	Description
sched	5,712	RS/6000 Instruction Timing Simulator
ixx	11,157	IDL specification to C++ stub-code translator
lcom	17,278	Compiler for the “L” hardware description language
hotwire	5,335	Scriptable graphical presentation builder
simulate	6,672	Simula-like simulation class library and example
idl	30,288	SunSoft IDL compiler with demo back end
taldict	11,854	Taligent dictionary benchmark
deltablue	1,250	Incremental dataflow constraint solver
richards	606	Simple operating system simulator

Table 5.1: Benchmark Programs. Size is given in non-blank lines of code.

5.4.2 Benchmarks

Table 5.1 describes the benchmarks we used in this study. Of the nine programs, we consider seven to be “real” programs (`sched, ixx, lcom, hotwire, simulate, idl` and `taldict`) which can be used to draw meaningful conclusions about how the analysis algorithms will perform. `idl` and `taldict` are both programs made up of production code with demo drivers; the rest are all programs used to solve real problems. The remaining two benchmarks, `richards` and `deltablue`, are included because they have been used in other papers and serve as a basis for comparison and validation.

Table 5.2 provides an overview of the static characteristics of the programs in absolute terms. Library code is not included. The number of functions, call sites, and virtual call arcs gives a composite picture of the static complexity of the program. Live call sites are those which were executed in our traces. Non-dead virtual call sites are those call sites, both resolved and unresolved, that remained in the program after our most aggressive analysis (RTA) removed some of the dead functions and the virtual call sites they contained.

Table 5.3 provides an overview of the dynamic (execution time) program characteristics for optimized code. Once again, all numbers are for user code only. The number of instructions between virtual function calls is an excellent (though crude) indication of how much potential there is for speedup from virtual function resolution. Under IBM's AIX operating system and C++ run-time environment a virtual function call takes 12 instructions, meaning that the user code of `taldict` could be sped up by a factor of two

Program	Code Size (bytes)	Fns	Call Sites	Live Sites	Virtual Sites		Virtual Instances
					Total	Live	
sched	99,888	237	530	184	34	33	58
ixx	178,636	1,108	3,601	767	467	399	1,752
lcom	164,032	779	2,794	1,653	458	446	3,661
hotwire	45,416	230	1,204	550	48	6	83
simulate	28,900	242	580	141	36	23	41
idl	243,748	856	3,671	882	1,248	1,198	3,486
taldict	20,516	429	783	47	79	14	116
deltablue	N.A.	103	372	201	3	3	11
richards	9,744	78	174	68	1	1	5

Table 5.2: Totals for static (compile-time) quantities measured in this chapter. All quantities are measured for user code only (libraries linked to the program are not included).

Program	Instrs. Executed	Function Calls [3]	Virtual Calls [5]	Instrs. betw. Virtual Calls
sched	106,901,207	2,302,003	967,789	110
ixx	7,919,945	248,391	47,138	168
lcom	107,826,169	4,210,059	1,099,317	98
hotwire	4,842,856	189,160	33,504	145
simulate	1,230,305	57,537	10,848	113
idl	776,792	33,826	14,211	55
taldict	837,496,535	39,401,445	35,060,980	23
deltablue	10,492,752	558,028	205,100	51
richards	86,916,173	2,407,782	657,900	132

Table 5.3: Totals for dynamic (run-time) quantities measured in this chapter. All quantities are for user code only (libraries linked to the program are not included).

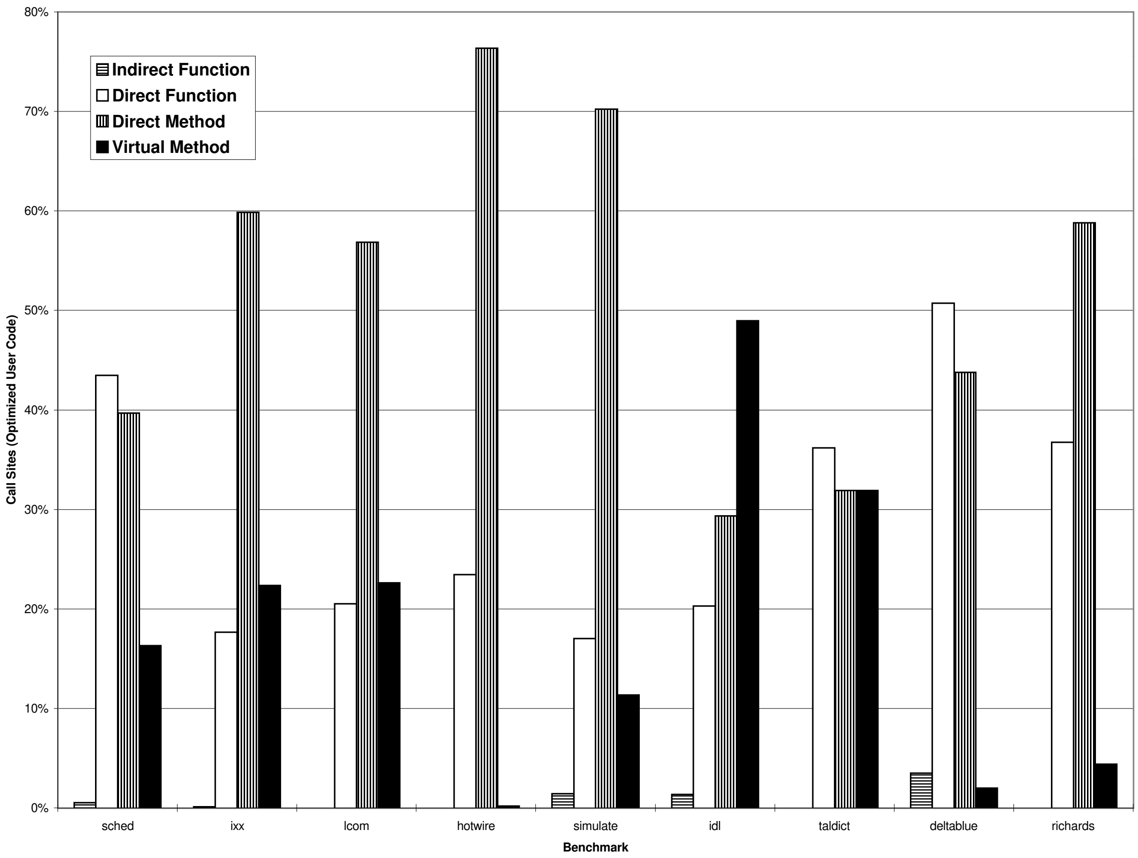


Figure 5.3: Static Distribution of Function Call Types

if all virtual calls are resolved (as they in fact are).

Our graphs all use percentages because the absolute numbers vary so much. Tables 5.2 and 5.3 include the totals for all subsequent graphs.

Figure 5.3 is a bar graph showing the distribution of types of live call sites contained in the user code of the programs; Figure 5.4 shows the analogous figures for the number of dynamic calls in user code. Direct (non-virtual) method calls account for an average of 51% of the static call sites in the seven large applications, but only 39% of the dynamic calls. Virtual method calls account for only 21% of the static call sites, but a much more significant 36% of the total dynamic calls.

Indirect function calls are used sparely except by `deltablue`, and pointer-to-member calls are only used by `ixx`, and then so infrequently that they do not appear on the bar chart.

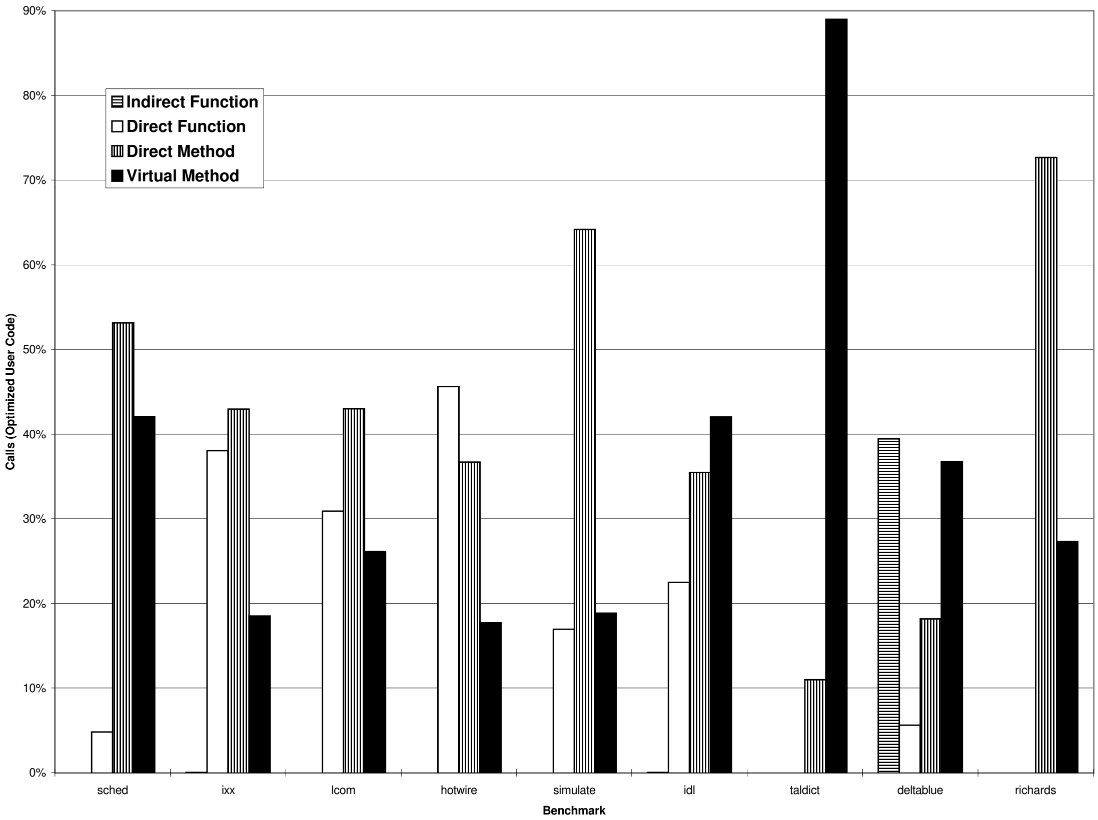


Figure 5.4: Dynamic Distribution of Function Call Types

Since non-virtual and virtual method calls are about evenly mixed, and direct (non-method) calls are less frequent, we conclude that the programs are written in a relatively object-oriented style. However, only some of the classes are implemented in a highly reusable fashion, because half of the method calls are non-virtual. The exception is `taldict`, with 89% of the dynamic function calls virtual: `taldict` uses the Taligent frameworks, which are designed to be highly re-usable. As use of C++ becomes more widespread and code reuse becomes more common, we expect that programs will become more like `taldict`, although probably not to such an extreme.

Note that we assume that trivially resolvable virtual function calls are implemented as direct calls, and count them accordingly throughout our measurements. That is, the call to `foo()` in

```
A a;
a.foo();
```

is considered a direct call even if `foo()` is a virtual function. This is consistent with the capabilities of current production C++ compilers, but different from the assumption used in some related work, in particular that of Pande and Ryder.

Our results differ, in some cases significantly, from those reported in two previous studies of C++ virtual function call resolution [Calder and Grunwald 1994; Aigner and Hölzle 1996]. This would seem to indicate that there is considerable variation among applications. Where possible we have used their benchmarks. However, many of their benchmarks are written in G++, the Gnu project's version of C++, which contains incompatible extensions to the language.

Considerable additional work remains to be done for benchmarking of C++ programs. While the SPEC benchmark suite has boiled down "representative" C code to a small number of programs, it may well be that such an approach will not work with C++ because it is a more diverse language with more diverse usage patterns.

5.4.3 Live Classes

The fundamental basis of the Rapid Type Analysis algorithm is its computation of a set of live classes, C_L. RTA is superior to Class Hierarchy Analysis because of its use of live class information.

Figure 5.5 shows the ability of RTA to find dead classes in our benchmarks, as compared to the total number of classes that were not created during our traces of the benchmarks. In this and most subsequent graphs, we use dynamic information to provide an upper bound on the performance of static analysis. Such an upper bound is always shown using a white bar.

The point of such "upper bound" measurements is to provide some information as to how well our RTA algorithm is doing against what could be achieved by the best possible static algorithm. Of course, since we are using a dynamic trace for the upper bound, all we know is that the actual best precision of a static analysis algorithm lies somewhere between the precision achieved by RTA and the hypothetical precision shown by using a dynamic trace.

If the difference between RTA and the dynamic trace is small, then the measurements have provided strong information: RTA has done very well, and there is not much room for improvement by other algorithms (as is the case for the benchmarks `sched`, `simulate`, `taldict`, and `deltablue` in Figure 5.5). If the difference between RTA and the dynamic

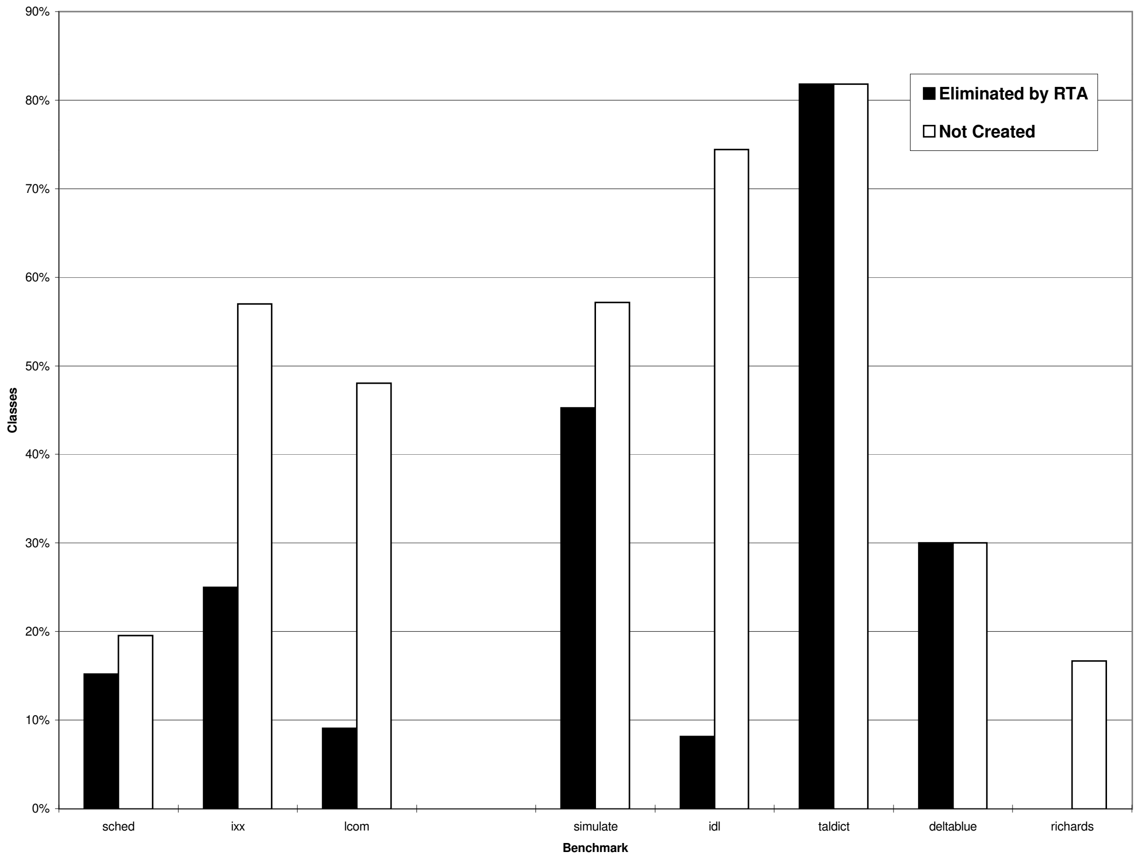


Figure 5.5: Unused Classes: RTA Algorithm vs. Dynamic Trace.

trace is large, then the measurements have only provided weak information: there may be room for improvement, or there may not (as is the case with the benchmarks `ixx`, `lcom`, `idl`, and `richards`).

Unsurprisingly, RTA is only 100% precise for two synthetic benchmarks, `taldict` and `deltablue`. In real programs, RTA is unable to create a precise call graph, and therefore the set of live classes is also imprecise. The only benchmark for which no dead classes were found is also the synthetic `richards` benchmark.

While it is difficult to predict what kind of performance impact the live class information will have, the measurements in Figure 5.5 do show some promise that RTA-based optimizations will be superior to those performed using more primitive analysis methods like CHA.

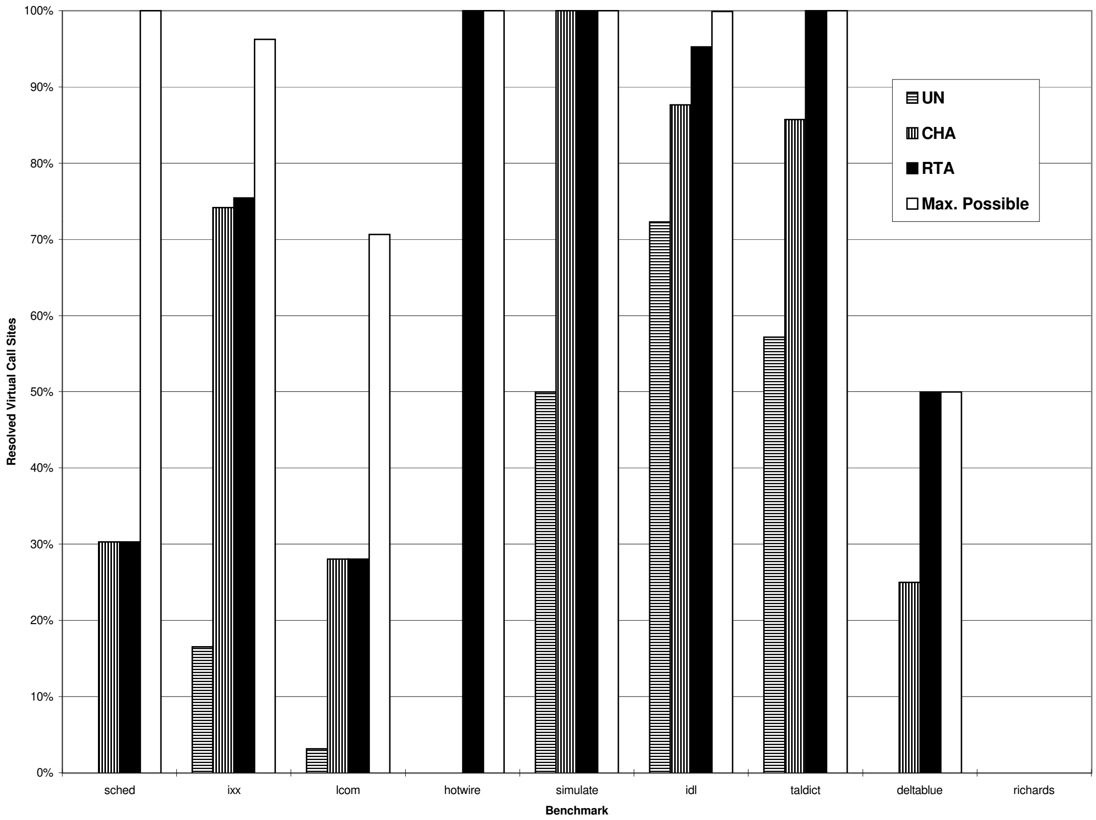


Figure 5.6: Resolution of Possibly Live Static Callsites

5.4.4 Resolution of Virtual Function Calls

When a virtual call site always calls the same function during one or more runs of the program, we say that it is *monomorphic*. If it calls multiple functions, it is *polymorphic*. If the optimizer can prove that a call that was monomorphic during a single execution will be monomorphic under *all* program executions, then the call can be resolved statically. Polymorphic call sites can not be resolved unless the enclosing code is cloned or type tests are inserted.

The performance of the analyses for resolving virtual function calls is shown in Figures 5.6 (which presents the static information for the call sites) and 5.7 (which presents the dynamic information for the calls in our program traces). Together with the remaining graphs they compare the performance of the three static analysis algorithms, and they all use a consistent labeling to aid in interpretation.

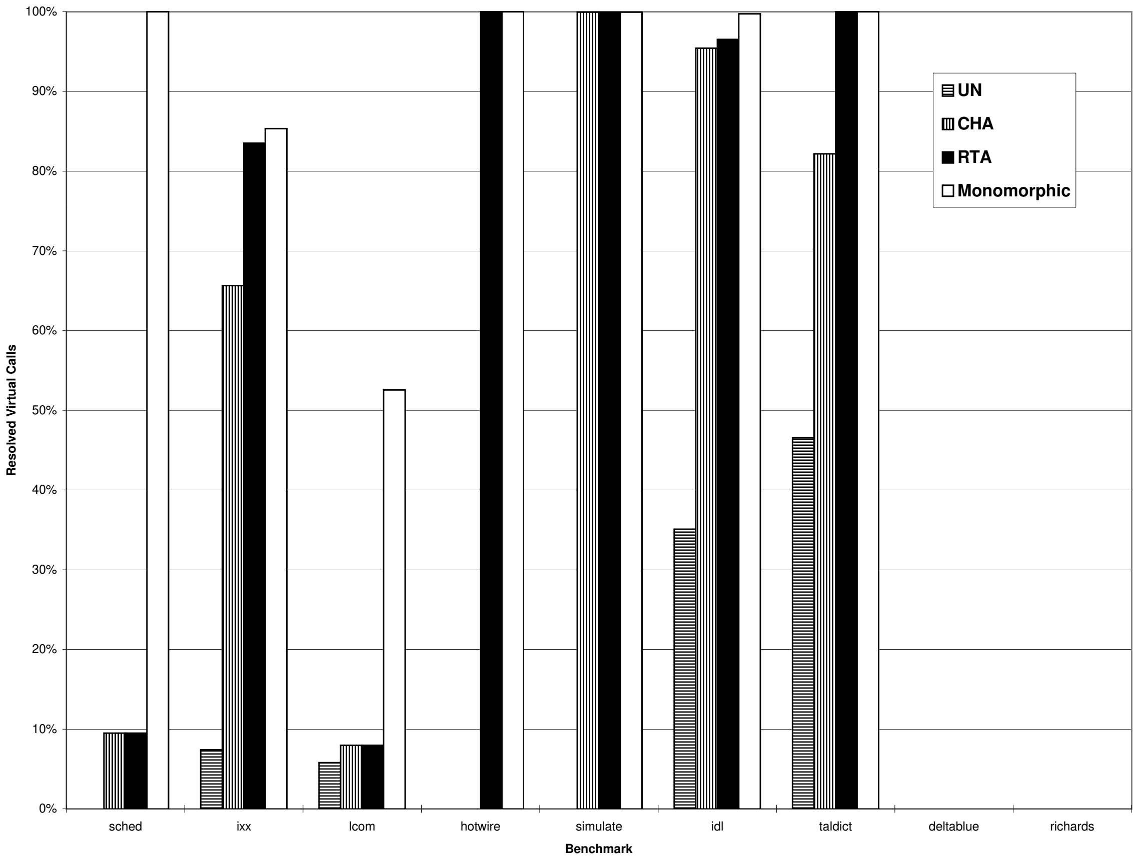


Figure 5.7: Resolution of Dynamic Calls

In the static measurements of Figure 5.6, the white bar (the "region of opportunity" for finer analysis) represents call sites that were either dynamically monomorphic, or were not executed during the trace but were not eliminated as dead code by RTA. In Figure 5.7, which measures dynamic calls, the white bar represents the monomorphic calls. The benchmarks `deltablue` and `richards` are fully polymorphic at run-time, so there are no bars for those benchmarks in the figure.

Call sites identified as dead by Rapid Type Analysis were not counted, regardless of whether they were resolved. This was done so that the static and dynamic measurements could be more meaningfully compared, and because it seemed pointless to count as resolved a call site in a function that can never be executed. However, this has relatively little effect on the overall percentages.

Figure 5.7 shows that for for five out of seven of the large benchmarks, the most powerful static analysis, RTA, resolves all or almost all of the virtual function calls. In other words, in five out of seven benchmarks, RTA does an essentially perfect job. On average, RTA resolves 71% of the dynamic virtual calls in the seven large benchmarks. CHA is also quite effective, resolving an average of 51%, while UN performs relatively poorly, resolving an average of 15% of the dynamic virtual calls.

We were surprised by the poor performance of Unique Name, since Calder and Grunwald found that Unique Name resolved an average of 32% of the virtual calls in their benchmarks. We are not sure why this should be so; possibly our benchmarks, being on average of a later vintage, contain more complex class hierarchies. UN relies on there only being a single function in the entire application with a particular signature.

Our benchmarks are surprisingly monomorphic; only two of the large applications (`ixx` and `lcom`) exhibit a significant degree of polymorphism. We do not expect this to be typical of C++ applications, but perhaps monomorphic code is more common than is generally believed.

A problem arose with one program, `lcom`, which is not type-safe: applying CHA or RTA generates some specious call site resolutions. We examined the program and found that many virtual calls were *potentially unsafe*, because the code used down-casts. However, most of these potentially unsafe calls *are in fact safe*, because the program uses a collection class defined to hold pointers of type `void*`. Usually, inspection of the code shows that the down-casts are simply being used to restore a `void*` pointer to the original type of the object inserted into the collection.

We therefore selectively turned off virtual function call resolution at the call sites that could not be determined to be safe; only 7% of the virtual calls that would have been resolved by static analysis were left unresolved because of this change (they are counted as unresolved monomorphic calls). We feel that this is a reasonable course because a programmer trying to optimize his or her own program might very well choose to follow this course rather than give up on optimization altogether; readers will have to use their own judgment as to whether this would be an acceptable programming practice in their environments. However, a better alternative in the case of `lcom` would simply be to use template for the collection classes, thereby avoiding the need for the down-casting operations.

The only benchmark to use library code containing virtual calls was `simulate`, which uses the task library supplied with AIX. Slightly less than half of the virtual calls were made from the library code, and about half of those calls were monomorphic (and therefore potentially resolvable). We have not included virtual calls in library code in the graphs because the corresponding code was not available to static analysis.

Why Rapid Type Analysis Wins

Since Class Hierarchy Analysis is a known and accepted method for fast virtual function resolution, it is important to understand why RTA is able to do better.

RTA does better on four of seven programs, although for `idl` the improvement is minor. For `ixx`, RTA resolves a small number of additional static call sites (barely visible in Figure 5.6), which account for almost 20% of the total dynamic virtual function calls. The reason is that those calls are all to frequently executed string operations. There is a base class `String` with a number of virtual methods, and a derived class `UniqueString`, which overrides those methods. RTA determines that no `UniqueString` objects are created in `ixx`, and so it is able to resolve the virtual call sites to `String` methods. These call sites are in inner loops, and therefore account for a disproportionate number of the dynamic virtual calls.

RTA also makes a significant difference for `taldict`, resolving the remaining 19% of unresolved virtual calls. RTA is able to resolve two additional call sites because they are calls where a hash table class is calling the method of an object used to compare key values. The comparison object base class provides a default comparison method, but the derived class used in `taldict` overrides it. RTA finds that no instances of the base class are created, so it is able to resolve the calls.

The `hotwire` benchmark is a perfect example of the class library scenario: a situation in which an application is built using only a small portion of the functionality of a class library. The application itself is a simple dynamic overhead transparency generator; it uses a library of window management and graphics routines. However, it only creates windows of the root type, which can display text in arbitrary fonts at arbitrary locations. All of the dynamic dispatch occurs on redisplay of sub-windows, of which there are none in this application. Therefore, all of the live virtual call sites are resolved.

Why Fast Static Analysis Fails

One benchmark, `sched`, stands out for the poor performance of all three static analysis algorithms evaluated in this chapter. Only 10% of the dynamic calls are resolved, even though 30% of the static call sites are resolved, and 100% of the dynamic calls are monomorphic. Of course, a function may be monomorphic with one input but not with another. However, `sched` appears to be completely monomorphic.

The unresolved monomorphic virtual call sites are all due to one particular programming idiom: `sched` defines a class `Base` and two derived classes `Derived1` and `Derived2` (not their real names). `Base` has no data members, and defines a number of virtual functions whose implementation is always `assert(false)` – in other words, they will raise an exception when executed. In essence, `Base` is a strange sort of abstract base class.

`Derived1` and `Derived2` each implement a mutually exclusive subset of the methods defined by `Base`, and since `Base` has no data members, this means that these two object types are totally disjoint in functionality. It is not clear why the common base class is being used at all.

RTA determines that no objects of type `Base` are ever created. However, the calls to the methods of `Derived1` and `Derived2` are always through pointers of type `Base*`. Therefore, there are always two possible implementations of each virtual function: the one defined by one of the derived classes, and the one inherited from `Base` by the other derived class.

Depending on your point of view, this is either an example of the inability of static analysis to handle particular coding styles, or another excellent reason not to write strange code.

The other benchmark for which none of the static analyses do a very good job is `lcom`: 45% of the virtual calls are monomorphic but unresolved. 40% of the virtual calls are from a single unresolved call site. These calls are all through an object passed in from a single procedure, further up in the call graph. That procedure creates the object with **new**, and it is always of the same type. While it would probably not be resolved by simple flow analysis, it could be resolved by alias analysis.

What kinds of programming idioms are not amenable to fast static analysis? CHA will resolve monomorphic virtual calls for which there is only a single possible target. RTA will also eliminate monomorphic calls when only one of the possible target object types is used in the program. The kind of monomorphic calls that can't be resolved by RTA occur when multiple related object types are used independently, for instance if `Square` and `Circle` objects were each kept on their own linked list, instead of being mixed together. This is known as *parametric polymorphism*.

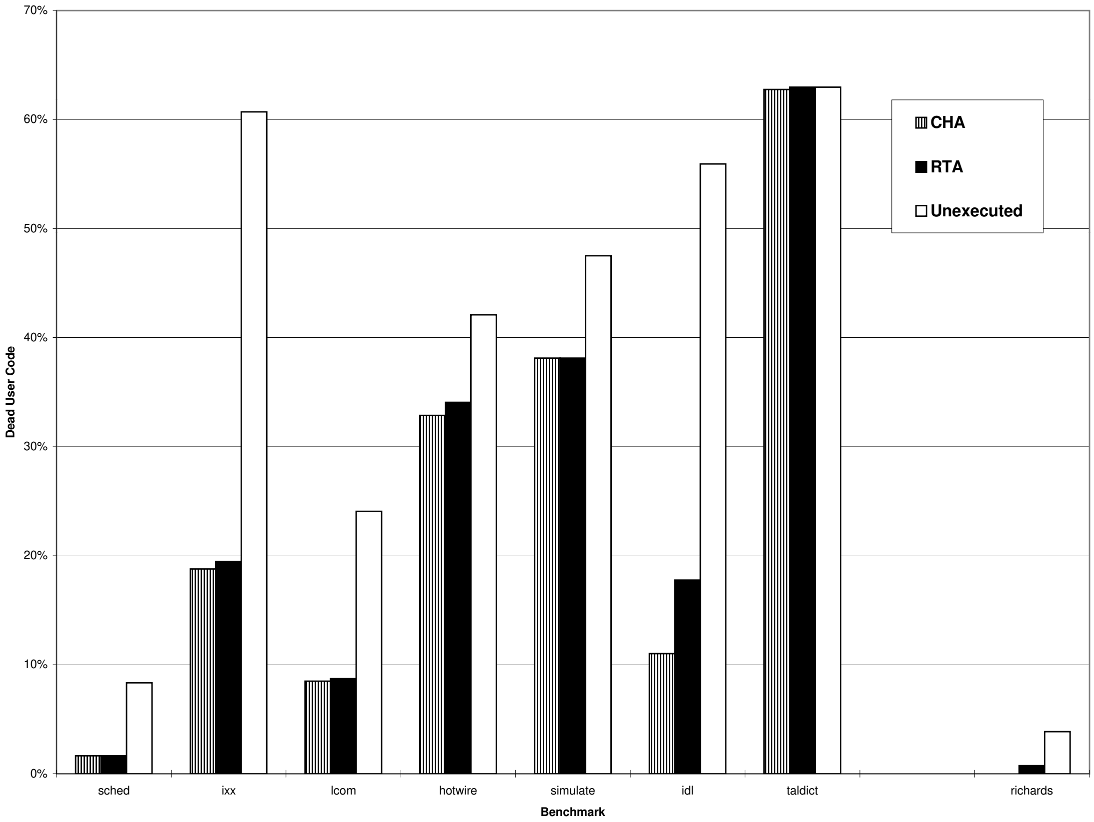


Figure 5.8: Removal of Dead Code by Static Analysis

Parametric polymorphism is what occurs in `lcom` and, in a degenerate fashion, in `sched`. Parametric polymorphism presents the major opportunity for alias analysis to improve upon the fast static techniques presented in this dissertation, since it can sometimes determine that a pointer can only point to one type of object even when multiple possible object types have been created.

5.4.5 Code Size

Because they build a call graph, Class Hierarchy Analysis and Rapid Type Analysis identify some functions as dead: those that are not reachable in the call graph. RTA is more precise because it removes virtual call arcs to methods of uninstantiated objects from the call graph.

Figure 5.8 shows the effect of static analysis on user code size. As before, white represents the region of opportunity for finer analysis – code that was not executed during the trace and might be dead or might be executed when the benchmark is run with a different input. Code size is measured in bytes of compiled object code.

Our measurements include only first-order effects of code size reduction due to the elimination of entire functions. There is a secondary code-size reduction caused by resolving virtual call sites, since calling sequences for direct calls are shorter than for virtual calls. We also did not measure potential code expansion (or contraction) caused by inlining of resolved call sites. Finally, due to technical problems our code size measurements are for unoptimized code, and we were not able to obtain measurements for `deltablue`.

On average, 42% of the code in the seven large benchmarks is not executed during our traces. Class Hierarchy Analysis eliminates an average of 24% of the code from these benchmarks, and Rapid Type Analysis gets about one percent more.

CHA and RTA do very well at reducing code size: in five of the seven large benchmarks, unexecuted code that was not removed by static analysis accounts for less than 20% of the total program size. Only `ixx` and `idl` contain significant portions of code that was neither executed nor eliminated (about 40%).

We were surprised to find that despite the fact that RTA does substantially better than CHA at virtual function resolution, it does not make much difference in reducing code size. Unique Name does not remove any functions because it only resolves virtual calls; it does not build a call graph.

5.4.6 Static Complexity

Another important advantage of static analysis is its use in programming environments and compilers. For instance, in presenting a user with a program browser, the task of understanding the program is significantly easier if large numbers of dead functions are not included, and if virtual functions that can not be reached are not included at virtual call sites.

In addition, the cost and precision of other forms of static analysis and optimization are improved when the call graph is smaller and less complex.

Figure 5.9 shows the effect of static analysis on eliminating functions from the call graph. This is similar to Figure 5.8, except that each function is weighted equally, instead of being weighted by the size of the compiled code. Therefore, Figure 5.9 shows the number of functions eliminated rather than the reduction in code size. As we stated above, since Unique Name does not build a call graph, it does not eliminate any functions.

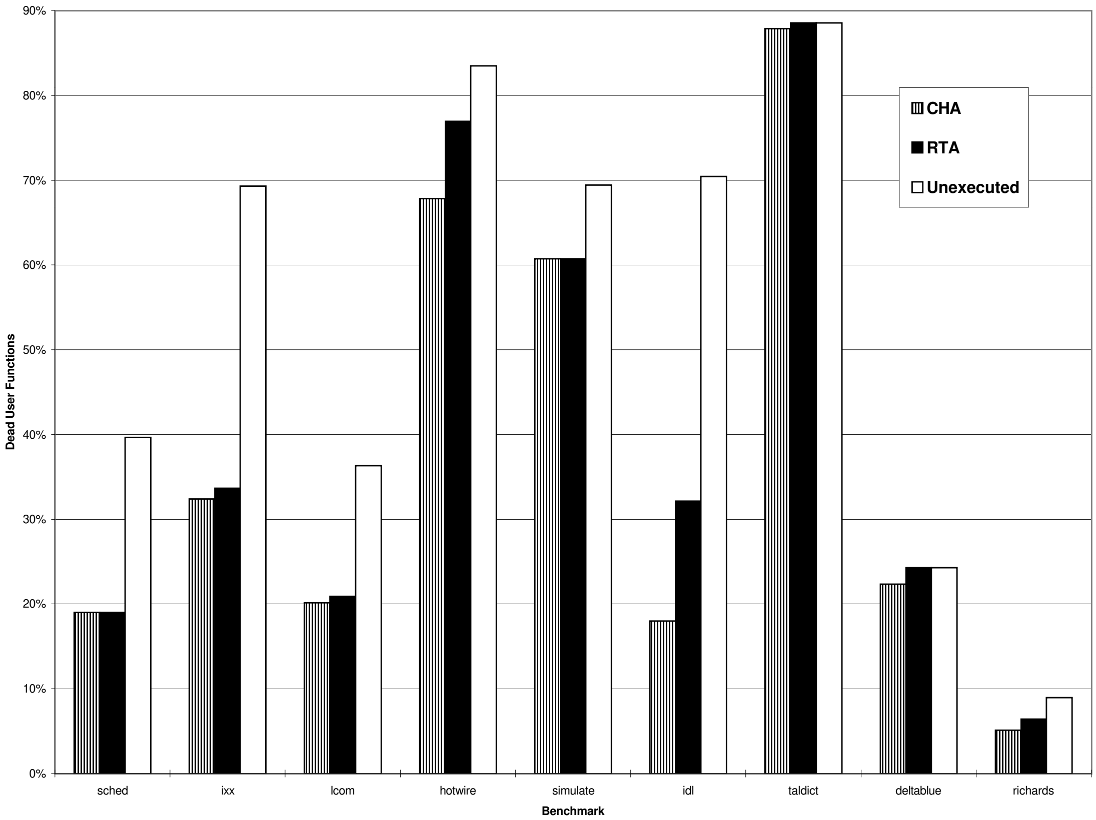


Figure 5.9: Elimination of Dead Functions by Static Analysis

Once again, Class Hierarchy Analysis eliminates a large number of functions, and Rapid Type Analysis eliminates a few more.

Figure 5.10 shows the effect of static analysis on the number of virtual call instances in the call graph. As defined in Section 3.4, a *call instance* is an arc in the call graph from a virtual call site to one of its potential dynamic targets.

Class Hierarchy Analysis removes call instances because it eliminates functions, and so any call instances that they contain are also removed. Rapid Type Analysis can both remove dead functions and remove virtual call instances in live functions. For example, refer back to Figure 4.2 at the beginning of this chapter: even though `main()` is a live function, RTA removes the call instance to `A::foo()` at the call that produces `result3` because it discovers that no objects of type `A` are ever created.

Surprisingly, despite the large number of virtual call sites that are resolved in most programs, relatively few virtual call instances are removed in three of the seven large

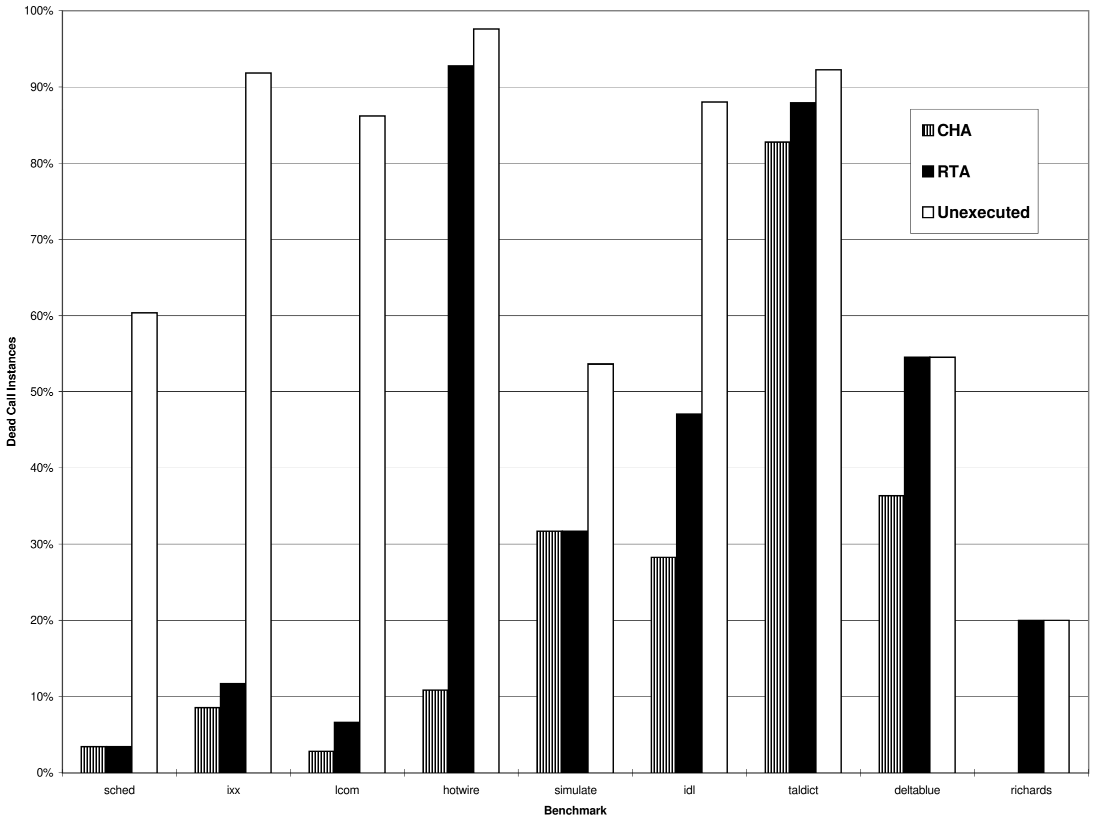


Figure 5.10: Elimination of Virtual Call Instances by Static Analysis

benchmarks. In those programs, the virtual function resolution is due mostly to Class Hierarchy Analysis. CHA, by definition, resolves a function call when there is statically only a single possible target function at the call site. Therefore, the call site is resolved, but the call instance is not removed. On the other hand, because RTA actually removes call instances in live functions, it may eliminate substantial numbers of call instances, as is seen in the case of `hotwire`.

5.4.7 Speed of Analysis

We have claimed that a major advantage of the algorithms described in this dissertation is their speed. Table 5.4 shows the cost of performing the Class Hierarchy Analysis and Rapid Type Analysis algorithms on an 80 MHz PowerPC 601, a modest CPU by today's standards. The total time to compile and link the program is also included for comparison. We do not include timings for Unique Name because we implemented it in a manner that

Benchmark	Size (lines)	Analysis Time CHA	Analysis Time RTA	Compile Time	RTA Overhead
sched	5,712	1.90	1.94	921	< 0.1%
ixx	11,157	5.12	5.22	367	1.4%
lcom	17,278	6.27	6.50	218	3.0%
hotwire	5,335	2.05	2.06	160	1.3%
simulate	6,672	2.67	2.75	49	5.6%
idl	30,288	5.71	6.42	450	1.4%
taldict	11,854	1.66	1.78	45	4.0%
deltablue	1,250	0.42	0.44	18	2.4%
richards	606	0.30	0.32	9	3.6%

Table 5.4: Compile-Time Cost of Static Analysis (timings are in seconds on an 80 MHz PowerPC 601). Compile time is for optimized code, and includes linking. Rightmost column shows the overhead of adding RTA to the compilation process.

maximized re-use of our other code, rather than optimizing the Unique Name algorithm itself. Since Unique Name performed poorly compared to CHA and RTA, we did not feel it was worth the extra effort of a "native" implementation.

RTA is not significantly more expensive than CHA. This is because the major cost for both algorithms is that of traversing the program and identifying all the call sites. Once this has been done, the actual analysis proceeds very quickly.

RTA analyzes an average of 3310 non-blank source lines per second, and CHA is only marginally faster. The entire 17,278-line `lcom` benchmark was analyzed in 6.5 seconds, which is only 3% of the time required to compile and link the code. On average, RTA took 2.4% of the total time to compile and link the program.

We expect that these timings could be improved significantly; our implementation is a prototype, designed primarily for correctness rather than speed. No optimization or tuning has been performed yet.

Even without improvement, 3300 lines per second is fast enough to include in a production compiler without significantly increasing compile times.

5.5 Related Work

5.5.1 Type Prediction for C++

Aigner and Hölzle [1996] compared the execution time performance improvements due to elimination of virtual function calls via class hierarchy analysis and profile-based type prediction. Our work differs from theirs in that we compare three different static analysis techniques, and in that we demonstrate the ability of static analysis to reduce code size and reduce program complexity. We also use dynamic information to bound the performance of static analysis.

Type prediction has advantages and disadvantages compared with static analysis. Its advantages are that it resolves more calls, and does not rely on the type-correctness of the program. Its disadvantages are that it requires the introduction of a run-time test; it requires profiling; and it is potentially dependent upon the input used during the profile.

Type prediction can always "resolve" more virtual calls than static analysis, because it precedes a direct call with a run-time test. Call sites resolved by static analysis do not need to perform this test, and one would therefore expect the execution time benefit from static resolution to be greater than that from type prediction. This trend is indeed evident in Aigner and Hölzle's execution time numbers: for only one of their benchmarks does type feedback provide more than a 3% speedup over Class Hierarchy Analysis. This is despite the fact that in all but one of the benchmarks, type prediction resolves a significantly larger number of virtual calls.

We performed a simple experiment to evaluate the benefits of type analysis versus type prediction. A program consisting of a loop that repeatedly calls a virtual function was timed, both without virtual function resolution, with static resolution, and with type prediction. The latter two were tested with and without inlining; the called function was trivial so that the effect of inlining is maximized. The results are shown in Figure 5.11.

At 100% accuracy, type prediction does quite well, although static resolution is still significantly faster. But as the accuracy of type prediction decreases, static resolution looks better and better. Type prediction can even worsen the performance of the program: without inlining, type prediction must achieve 54% accuracy to break even, and with inlining it must achieve 37% accuracy.

However, computer science is a field in which it pays to play the odds. Type prediction often achieves accuracy close to 100%. Ultimately, we believe that a combination of static

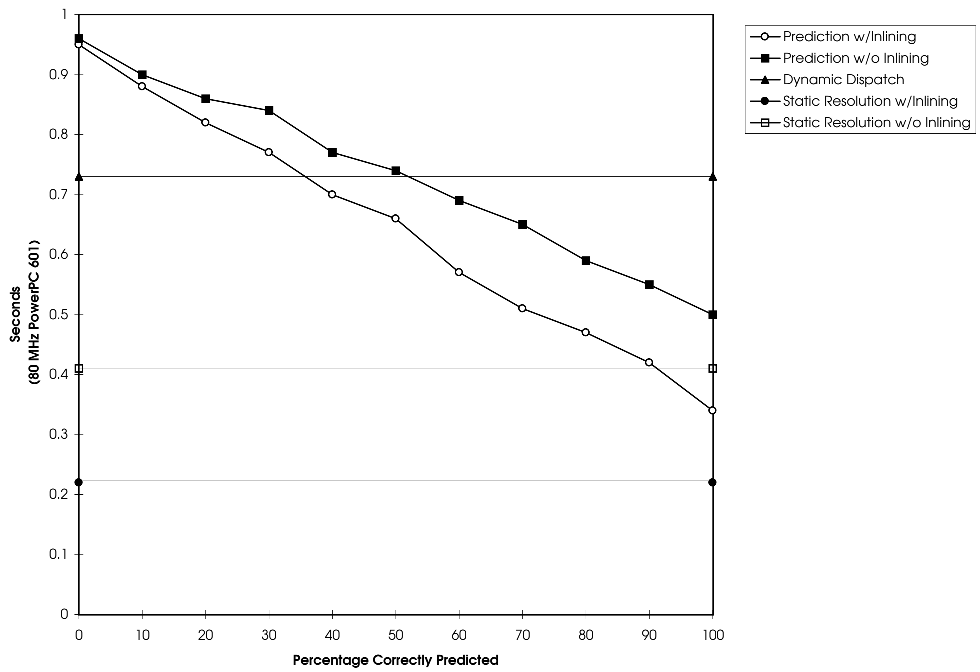


Figure 5.11: Comparison of Type Prediction vs. Virtual Function Resolution

analysis with type prediction is likely to be the best solution. But between the two, static resolution is preferable both because of its superior performance and because it is not subject to degradation with different inputs.

In Aigner and Hölzle's study, excluding the trivial benchmarks `deltablue` and `richards` and weighting each program equally, Class Hierarchy Analysis resolved an average of 27% of the dynamic virtual function calls (and a median of 9%). Aigner and Hölzle said they were surprised by the poor performance of CHA on their benchmarks, since others had found it to perform well. In our measurements, CHA resolved an average of 51% of the dynamic virtual calls, so it seems that there is considerable variation depending upon the benchmark suite. In fact, we got different results for the one large benchmark that we had in common, `ixx`, due to a different input file and possibly a different version of the program.

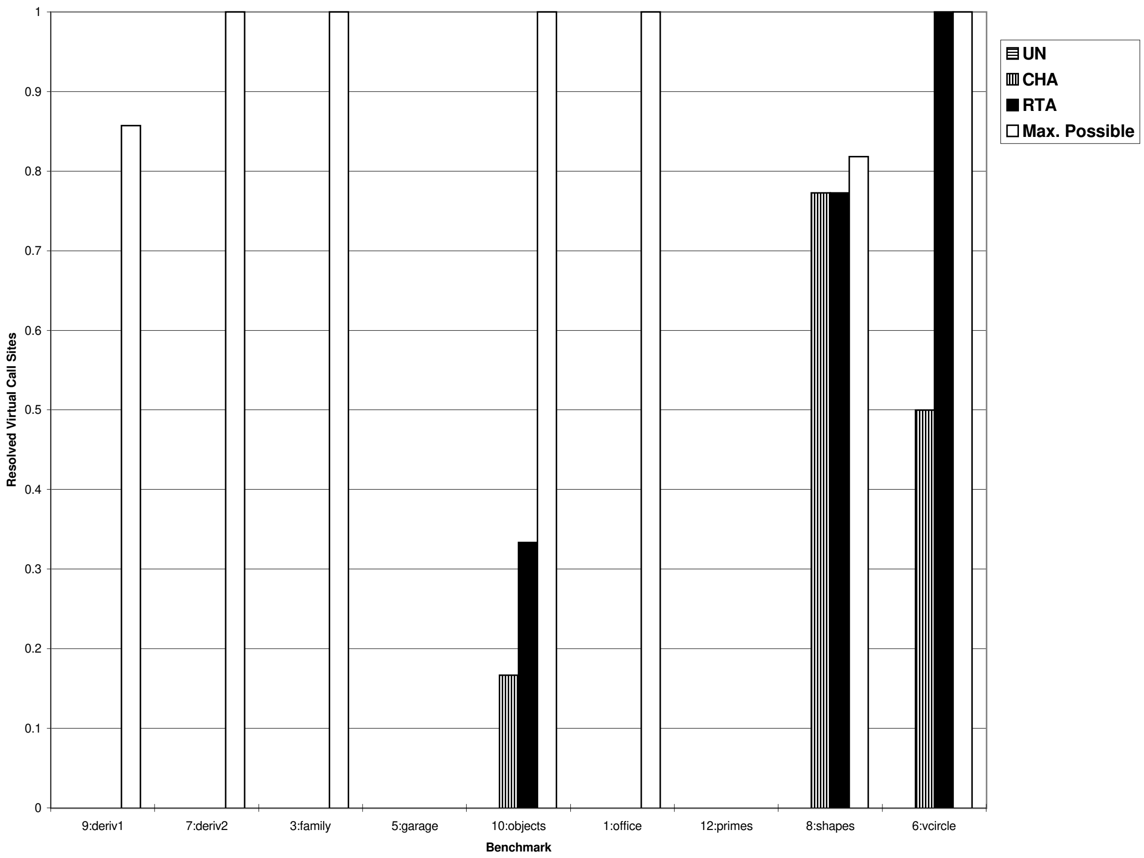


Figure 5.12: Resolution of Static Callsites – Alias Analysis Benchmarks

5.5.2 Alias Analysis for C++

The most precise, and also most expensive, proposed static method for resolving virtual function calls is to use interprocedural flow-sensitive alias analysis. Pande and Ryder [1996; 1994] have implemented an alias analysis algorithm for C++ based on Landi et al.'s [1993] algorithm for C. This analysis is then used to drive virtual function elimination. They give preliminary results for a set of 19 benchmark programs, ranging in size from 31 to 968 lines of code.

In comparison with our RTA algorithm, which processes about 3300 lines of source code per second (on an 80 MHz PowerPC 601), the speed of their algorithm ranges from 0.4 to 55 lines of source code per second (on a Sparc-10). At this speed, alias analysis will not be practical in any normal compilation path.

We have obtained their benchmark suite; Figure 5.12 shows the performance of our static analysis algorithms on the 9 programs that we could execute (their analysis is purely static, and not all of their programs were actually executable). Of these 9, two are completely polymorphic (no resolution is possible), and two were completely or almost completely resolved by Rapid Type Analysis or Class Hierarchy Analysis. So for four out of nine, RTA does as well as alias analysis.

RTA resolved 33% of the virtual call sites in `objects`, compared to about 50% by alias analysis (for comparative data, see their paper [Pande and Ryder 1996]). For the remaining four (`deriv1, deriv2, family`, and `office`) fast static analysis did not resolve any virtual call sites, and significant fractions of the call sites were dynamically monomorphic. Alias analysis was able to resolve some of the virtual call sites in `deriv1` and `deriv2`, and all of the virtual call sites in `family` and `office`. However, the latter two programs are contrived examples where aliases are deliberately introduced to objects created in the `main` routine.

Because of the small size and unrealistic nature of the benchmarks used by Pande and Ryder, we hesitate to make any generalizations based on the results of our comparison. Two of our seven large benchmarks, `sched` and `lcom`, appear to be programs for which alias analysis could perform better than RTA. These programs make use of *parametric polymorphism*, as discussed in Section 5.4.4. We expect programs embodying this programming style to be the main beneficiaries of alias analysis, as applied to the virtual function resolution problem.

Over all, our benchmarks and Pande and Ryder's indicate that for most programs, there is relatively little room for improvement by alias analysis over RTA. However, there are definitely cases where alias analysis will make a significant difference. The ideal solution would be to use RTA first, and only employ alias analysis when RTA fails to resolve a large number of monomorphic calls.

Like Pande and Ryder, Carini et al. [1995] have devised an alias analysis algorithm for C++ based on an algorithm for C and Fortran [Choi et al. 1993; Burke et al. 1994]. We are currently collaborating with them on an implementation of their algorithm within our analysis framework. This will allow a direct comparison of both the precision and the efficiency of alias analysis.

5.5.3 Other Work in C++

Porat et al. [1996] implemented the Unique Name optimization in combination with type prediction in the IBM `xlC` compiler for AIX, and evaluated the results for 3 benchmark programs. Their two large benchmarks were identical to two of ours: `taldict` and `lcom`. They achieved a speedup of 1.25 on `taldict` and a speedup of 1.04 on `lcom`, using a combination of Unique Name and type prediction. Our estimates and experiments indicate that a significantly higher speedup is achievable for `taldict` using Rapid Type Analysis.

Calder and Grunwald [1994] implemented the first virtual function resolution algorithm for C++. Their Unique Name algorithm (which might more accurately be called "Unique Signature") is very fast, since it only requires a linear scan over the method declarations in the program. Calder and Grunwald implemented Unique Name as a link-time analysis, and found it to be quite effective. With their benchmarks, it resolved anywhere from 2.9% to 70.3% of the virtual calls executed by the program. We found it to be not nearly so effective on our benchmarks, and it was significantly outperformed by Rapid Type Analysis.

Srivastava [1992] developed an analysis technique with the sole purpose of eliminating unused procedures from C++ programs. He builds a graph starting at the root of the call graph. Virtual call sites are ignored; instead, when a constructor is reached, the referenced virtual methods of the corresponding class are added to the graph. His algorithm could also be used to resolve virtual function calls by eliminating uninstantiated classes from consideration and then using Class Hierarchy Analysis. His technique is less general than RTA because the resulting graph is not a true call graph, and can not be used as a basis for further optimization.

5.5.4 Other Related Work

Related work has been done in the context of other object-oriented languages such as Smalltalk, SELF, Cecil, and Modula-3. Of those, Modula-3 is the most similar to C++.

Fernandez [1995] implemented virtual function call elimination as part of her study on reducing the cost of opaque types in Modula-3. She essentially implemented Class Hierarchy Analysis, although only for the purpose of resolving virtual calls, and not for eliminating dead code.

Diwan et al. [1996] have investigated a number of algorithms for Modula-3, including an interprocedural uni-directional flow-sensitive technique, and a "name-sensitive" technique.

For the benchmarks they studied, their more powerful techniques were of significant benefit for Modula-3, because they eliminated the `NULL` class as a possible target. However, when `NULL` is ignored (as it is in C++), in all but one case the more sophisticated analyses did no better than Class Hierarchy Analysis. This is interesting because we found several cases in which Rapid Type Analysis was significantly better than Class Hierarchy Analysis – this may indicate that class instantiation information is more important than the flow-based information.

Because of the wide variation we have seen even among our C++ benchmarks, it seems unwise to extrapolate from Modula-3 results to C++. However, despite the difference between their and our algorithms, the basic conclusion is the same: that fast static analysis is very effective for statically typed object-oriented languages.

Dean et al. [1995] studied virtual method call elimination for the pure object-oriented language Cecil, which includes support for multi-methods. They analyzed the class hierarchy as we do to determine the set of type-correct targets of a virtual method call, and used this information to drive an intraprocedural flow analysis of the methods. Their method is not directly comparable to RTA: it uses more precise information within procedures, but performs no interprocedural analysis at all. Measured speedups for benchmarks of significant size were on the order of 25%, and code size reduction was also on the order of 25%.

There has been considerable work on type inference for dynamically typed languages [Plevyak and Chien 1994; Chambers and Ungar 1991a; Agesen 1994; Oxhøj et al. 1992]. In a recent paper, Agesen and Hölzle [1995] showed that type inference can do as well or better than dynamic receiver prediction in the SELF compiler, and proceeded to extrapolate from these results to C++ by excluding dispatches for control structures and primitive types. However, C++ and SELF may not be sufficiently similar for such comparisons to be meaningful.

5.5.5 Comparison of Available Algorithms

Now that we have described our experimental results and the salient characteristics of the various possible algorithms for solving the type analysis problem, we can return to the hypothesis put forward at the beginning of Chapter 4. That hypothesis, which is central

Algorithm	CH	DCE	Var	PPt	"Fast"
Flow-Sensitive Alias Analysis	•	•	•	•	
Flow-Insensitive Alias Analysis	•	•	•	•	
Single-set Alias Analysis	•	•	•		
Simple Type Propagation	•	•	•		
Rapid Type Analysis	•	•			•
Class Hierarchy Analysis	•				•
No Override	•				•
Unique Name					•
Unique Identifier					•

Figure 5.13: The Spectrum of Virtual Function Elimination Algorithms for Statically Typed Languages. We have implemented and compared the techniques listed in **bold**. CH=Builds class hierarchy; DCE=dead code elimination; Var=maintains per-variable information; PPt=maintains per-program-point information; Fast=algorithm expected to have minimal cost relative to total compile-time.

to this dissertation, is that *an algorithm exists for the type analysis problem which is close to optimal in precision, but is not very expensive to implement or execute.*

Conceptually, we illustrated this as a hypothesis that in the spectrum of algorithms, there is a "knee" in the time cost versus effectiveness curve of algorithms, and that there is an algorithm – specifically, Rapid Type Analysis – that is essentially at that inflection point (see Figure 4.1).

Our experimental results provide a strong indication that RTA usually comes close to optimal peformance in terms of the number of resolved calls. In other words, RTA has good "height" on the time/effectiveness curve. We have also shown, by implementing them in the same framework, that the Unique Name and Class Hierarchy Analysis algorithms are not a great deal faster than RTA.

However, what indication is there that there is not another algorithm that is not much slower than RTA, and which can improve on its effectiveness in a significant way? Our measurements indicate that for our set of benchmarks, there are two benchmarks for which significant improvements might be possible, leading to an improvement over all seven large benchmarks of about 10% in the number of resolved dynamic virtual calls.

Figure 5.13 compares a number of algorithms for the type analysis problem, in decreasing order of complexity, and (except for some variants of type propagation) in monotonicly decreasing order of precision. Dean et al. and Diwan have both implemented variants of type propagation.

What the figure shows is that the algorithms that are potentially more effective than RTA all have to keep per-variable information about the program, and have to examine each line of code for its potential impact on those variables. In contrast, RTA only has to examine the call sites and only has to keep a few global sets of information.

While this is not absolutely conclusive, it is a very good indication that more precise algorithms are likely to be at least one or two orders of magnitude more expensive.

Therefore, we are quite confident that the time/effectiveness curve has an inflection point, and that RTA is at or very close to that "knee" in the curve.

5.6 Summary

We have investigated the ability of three types of static analysis to improve C++ programs by resolving virtual function calls, reducing compiled code size, and reducing program complexity to improve both human and automated program understanding and analysis.

We have shown that Rapid Type Analysis is highly effective for all of these purposes, and is also very fast. This combination of effectiveness and speed make Rapid Type Analysis an excellent candidate for inclusion in production C++ compilers.

RTA resolved an average of 71% of the virtual function calls in our benchmarks, and ran at an average speed of 3300 non-blank source lines per second. CHA resolved an average of 51% and UN resolved an average of only 15% of the virtual calls. CHA and RTA were essentially identical for reducing code size; UN is not designed to find dead code. RTA was significantly better than CHA at removing virtual call targets.

Unique Name was shown to be relatively ineffective, and can therefore not be recommended. Both RTA and CHA were quite effective. In some cases there was little difference, in other cases RTA performed substantially better. Because the cost of RTA in both compile-time and implementation complexity is almost identical to that of CHA, RTA is clearly the best of the three algorithms.

We have also shown, using dynamic traces, that the best fast static analysis (RTA) often resolves all or almost all of the virtual function calls (in five out of the seven large benchmarks). For these programs, there is no advantage to be gained by using more expensive static analysis algorithms like flow-sensitive type analysis or alias analysis. Since these algorithms will invariably be at least one to two orders of magnitude more expensive than RTA, RTA should be used first to reduce the complexity of the program and to determine if there are significant numbers of virtual call sites left to resolve. In some cases, this will allow the expensive analysis to be skipped altogether.

Chapter 6

De-virtualization of Inheritance

Multiple inheritance allows one class to be inherited more than once by a derived class. There are two possible meanings to ascribe to multiple inheritance: either there are multiple, distinct copies of the inherited class, or there is only one copy which is shared. In C++, the latter is distinguished by declaring the inheritance arc as `virtual`.

Other languages have attempted to cope with the language-design problems raised by multiple inheritance (or its omission) in varying ways. Java allows multiple inheritance of interfaces but not of implementation; Smalltalk allows only single inheritance.

Whether virtual inheritance is an option or the default for a language, it creates a problem in both the time and space dimensions of program performance. Because there are multiple paths to a base object, simple offsets can no longer be used for member access or function table lookup, which slows down these operations. In addition, the extra pointers required may cause the object size to increase significantly.

In one benchmark that used virtual bases in a significant manner, over 50% of the total object space allocated by the program was consumed by various pointers required by the object model [Sweeney 1997]. The *object model* is the set of design choices about how objects are represented in the system, including the way in which virtual function tables are layed out, the way in which the virtual inheritance relationships between portions of an object are represented, and the way in which the run-time type of the object is represented. The space overhead of virtual inheritance can be reduced by using offsets in the class information instead of pointers inside the object, but this in turn further slows down member access.

Therefore it is highly desirable to be able to convert virtual inheritance to non-virtual inheritance when the virtual inheritance can be shown to be unnecessary. Let us now discuss when such a situation would arise.

6.1 Uses of Virtual Inheritance

Virtual inheritance is almost always used when there is a possibility that the base class will be multiply inherited, and in particular, when classes are being defined as interfaces or as "mix-ins". A mix-in is a class that is not instantiated on its own, but is typically added to another class to extend its functionality. An example is a `ScrollBar` class, which can be mixed in with a `TextWindow` or a `GraphicsWindow` class to form a scroll-able window.

In designing system software, where it is difficult to predict how classes will be combined by application developers, there is a strong motivation to use virtual inheritance as a matter of course. However, virtual inheritance can cost additional object space (for the virtual base pointers) and additional time (because accessing base class members now requires an additional indirection).

From the standpoint of performance, virtual inheritance should only be used when a class will actually be multiply inherited, but how application developers will choose to inherit classes is difficult to predict. Once again, C++ confronts developers with a flexibility versus performance tradeoff.

However, when the complete application is being compiled, the extent of sub-classing is known. By analyzing the class hierarchy, the compiler can identify those cases for which virtual inheritance is actually needed. All other virtual inheritance edges can be marked as *pseudo-virtual*, meaning that for purposes of object layout, they can be implemented like non-virtual inheritance edges. The set of live classes C_L computed by Rapid Type Analysis (or alias analysis, or some other algorithm) can be used to identify classes that are never instantiated, thereby allowing a greater number of virtual inheritance edges to be eliminated.

Unfortunately, C++ requires a semantic difference between the manner in which virtual base class constructors and non-virtual base class constructors are called, so pseudo-virtual bases must still use virtual base constructor invocation semantics (a *virtual base class* of some derived class is a base class that is reached from the derived class through at least one virtual inheritance edge in the CHG). The practical effect of this difference in constructor semantics is that base class de-virtualization cannot be performed as a source-to-source

1 **findLiveCHG**(C, D, V, C_L)
2 $\quad C_\Lambda \leftarrow C_\Omega \leftarrow \emptyset$
3 $\quad$ **for each** $c \in C$
4 $\quad\quad mark(c) \leftarrow$ **false**
5 $\quad$ **for each** $c \in C : \nexists_{b \in C} : < b, c > \in D$
6 $\quad\quad$ **findLiveClasses**(c)

7 **findLiveClasses**$(c \in C)$
8 $\quad$ **if** $mark(c)$
9 $\quad\quad$ **return** $c \in C_\Lambda$
10 $\quad mark(c) \leftarrow$ **true**
11 $\quad l \leftarrow$ **false**
12 $\quad$ **for each** $d \in C : < c, d > \in D$
13 $\quad\quad l \leftarrow l$ **or findLiveClasses**(d)
14 $\quad$ **if** $c \in C_L$ **or** l
15 $\quad\quad C_\Lambda \leftarrow C_\Lambda \cup \{c\}$
16 $\quad$ **if** $c \in C_L$ **and not** l
17 $\quad\quad C_\Omega \leftarrow C_\Omega \cup \{c\}$
18 $\quad\quad$ **return true**
19 $\quad$ **else**
20 $\quad\quad$ **return** l

Figure 6.1: Finding the Live Portion of the CHG

translation, and requires additional support in code generation and the back end.

6.2 The Devirtualization Algorithm

To find devirtualizable inheritance edges in the class hierarchy, the compiler must first identify those classes that are relevant. Clearly, this must include the live classes (C_L), but it must also include any base classes of those live classes, since the compiler may be devirtualizing inheritance edges between non-live classes that are bases of live classes.

Figure 6.1 is a simple algorithm for finding this subgraph by using a depth-first traversal of the CHG. It computes two sets: C_Λ, the reduced class hierarchy induced by C_L, and $C_\Omega \subseteq C_L$, the set of leaf classes of C_Λ. Assuming that the set of roots of D that are iterated over on line 5 have been computed in advance, the **findLiveCHG** algorithm takes $O(D \log C)$ worst-case time or $O(D)$ expected time.

```
devirtualizeBases(C, D, V, C_Ω)
    for each d ∈ D
        if IsVirtual(d)
            mustBeVirtual(d) = false
    for each c ∈ C_Ω
        Q ← ∅
        initializeCounts(c)
        Q ← ∅
        findVirtualBases(c)
        Q ← ∅
        markVirtualEdges(c)

initializeCounts(c ∈ C)
    if c ∈ Q
        return
    Q ← Q ∪ {c}
    virtualCount(c) ← 0
    for each b ∈ C : < b, c >∈ D
        initializeCounts(b)

findVirtualBases(c ∈ C)
    if c ∈ Q
        return
    Q ← Q ∪ {c}
    for each b ∈ C : < b, c >∈ D
        if IsVirtual(< b, c >)
            virtualCount(b) ← virtualCount(b) + 1
        findVirtualBases(b)

markVirtualEdges(c ∈ C)
    if c ∈ Q
        return
    Q ← Q ∪ {c}
    for each b ∈ C : < b, c >∈ D
        if virtualCount(b) > 1
            mustBeVirtual(< b, c >) = true
        markVirtualEdges(b)
```

Figure 6.2: The Base Class De-virtualization Algorithm

Once the live class hierarchy has been computed, the algorithm **devirtualizeBases** in Figure 6.2 is relatively straightforward. Initially, all inheritance edges that were declared virtual are assumed to be implementable as non-virtual edges (lines 2–4). Then three passes are made up the class hierarchy from the live leaves in C_Ω. The purpose of these passes is to determine whether there are multiple paths with virtual edges from any live leaf to any of its bases.

The first pass, **initializeCounts**, simply initializes the counter *VirtualCount* to zero for all of the transitive base classes of the live leaf class c. The set Q is used to keep track of the visited classes to ensure that no class is visited more than once.

The second pass, **findVirtualBases**, walks up the class hierarchy from live leaf c and increments *VirtualCount* for every base class that is inherited virtually. After the second pass, any transitive base class x of leaf class c for which there is more than one virtual edge out of x that leads to c has a *VirtualCount* that is greater than 1.

The third pass, **markVirtualEdges**, walks up the class hierarchy and sets the *MustBeVirtual* flag of any virtual edges whose source class has a *VirtualCount* greater than 1.

6.3 Complexity

Each of the three passes traverses the class hierarchy in an identical manner, so we can simply analyze the complexity of **findVirtualBases** to find the complexity for all three (the only differences between them are in the constant-time operations of setting flags and so on).

Due to the use of the set Q, no edge of D will be traversed more than once per live leaf class, so there will be at most D recursive calls to **findVirtualBases**. The iterations in the loop on lines 23–26 all result in recursive calls, so those iterations must not be counted again. The set lookup and union operations for Q cost $\log C_\Lambda$, because by definition $Q \subseteq C_\Lambda$.

Therefore, the total cost of each pass is $O(D \log C_\Lambda)$ and the cost for the **devirtualizeBases** algorithm is

$$O(C_\Omega D \log C_\Lambda)$$

since the three passes are performed once for each live leaf class in C_Ω.

By using a hash table for Q, the expected time is reduced to $O(C_\Omega D)$. Since multiple inheritance is usually rare, in practice the cost of **devirtualizeBases** will usually be proportional to the size of D.

6.4 Evaluation

The algorithm described in this chapter can devirtualize a base class when it is not multiply inherited by any live classes. This allows class hierarchy designers to provide a lot of flexibility by declaring inheritance as virtual, but allows users of the class hierarchies to avoid the penalties of virtual inheritance unless it is truly necessary.

In the paradigm presented in the introduction to this dissertation, the base class devirtualization algorithm removes excess generality in the class hierarchy with respect to a particular program.

We were only able to find one benchmark of significant size that made non-trivial use of virtual base classes. Therefore, we have not done any quantitative evaluation, since one benchmark is insufficient for drawing any conclusions.

After searching fairly extensively for such programs, our impression is that the "lore" in the C++ community is that virtual base classes are too expensive. For instance, the Taligent frameworks, which would have been very naturally designed with virtual base classes, avoid them and were compromised in their orthogonality because the designers feared the performance implications.

While the base class devirtualization algorithm might or might not yield significant improvements for existing programs, if it is incorporated into compilers it will allow class hierarchy designers to be further freed from the need to trade off flexibility against performance.

Chapter 7

Other Uses of Live Class Information

In this chapter we will discuss some other optimizations that can be performed using the live class information that is computed by the Rapid Type Analysis algorithm, or by any other algorithm that computes live classes.

Primarily the optimizations described in this chapter apply to the features that in C++ are called "run-time type identification": the ability at run-time to inquire about the type of an object or cast an object to a derived type (or generate an exception if there is a type mismatch). These features are part of the new ANSI C++ standard (and therefore not much in use in C++), but are integral to Java because Java has no templates and encourages the use of generic container types.

The optimizations of this chapter involve a class c and an expression. We assume that RTA or some other static analysis provides a set E corresponding to the possible types that the expression could have at run-time. If the static type of the expression is $e \in C$, then RTA will compute the set

$$E = \mathit{Derived}^*(e) \cap C_L.$$

In other words, the set of possible dynamic types is the set of all live classes at or below e in the class hierarchy.

If an analysis algorithm that computes per-variable type information is used, such as alias analysis, then the set E will depend on the particular expression. However, the optimizations described in this chapter will be unaffected.

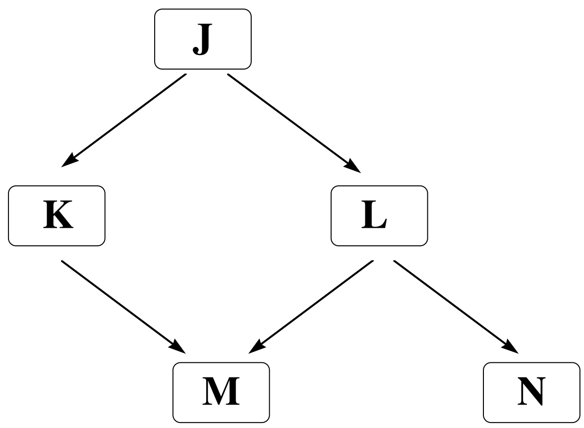


Figure 7.1: A Class Hierarchy that Complicates Type Inquiries

7.1 Optimizing Type Equality Tests

The simplest run-time type operation is to ask whether an object is of a particular type. C++ provides the **typeid** operator for this purpose. Assuming the class hierarchy in Figure 7.1, if we declare

```
K* p = new M();
```

then the expression

```
typeid(*p)==typeid(M)
```

will evaluate to `true`.

Using the set E of possible dynamic types of p calculated by RTA or some other analysis algorithm, if $E = \{\texttt{M}\}$ then we can replace `typeid(p)` by `typeid(M)`. The equality comparison can then be folded into a constant `true`. Similarly, if there is only one element in E but it is not `M`, then that type can be inserted and the expression will be folded into the constant **false**.

In general, if the set E contains more than one class, then a C++ **typeid** expression can not be evaluated at compile-time. However, the type expression above can still be optimized if $\texttt{M} \notin E$.

7.2 Optimizing Type Instance Tests

While C++ provides a type equality operator, Java provides a type instance operator, **instanceof**, whose semantics are more complex. Using the same class hierarchy from Figure 7.1 and assuming that J and L are interface classes, if we have the Java declaration

```
K p = new M();
```

then the expression

```
p instanceof M
```

will evaluate to true, just as with the C++ **typeid** operator. However, the expression

```
p instanceof L
```

will also evaluate to true, because `p` refers to an object of type `M`, and every `M` is an instance of an object of type `L`, its base class.

This property of the **instanceof** operator makes optimization considerably more involved, since not only do we have to contend with multiple possible types, but some of them may be siblings in the class hierarchy, rather than just ancestors and descendants (bases and derived classes). Therefore, for each class in $z \in E$, we must consider all of the classes in $Bases^*(z)$, the ancestors of z, when deciding whether the type inquiry expression can be optimized.

For the type instance inquiry `p instanceof L`, if $E = \{\mathtt{M}, \mathtt{N}\}$ then the expression can be optimized to the constant **true**. If $E = \{\mathtt{K}\}$ then the expression can be optimized to the constant **false**. If $E = \{\mathtt{K}, \mathtt{M}\}$ then the expression can not be optimized to a constant value, but it could be optimized to a simple test.

The algorithm for optimizing a type inquiry is shown in Figure 7.2. The function **optimizeInstanceOf** takes a set E of possible types for the expression and a class c, and returns **true** if the types in E are always instances of c, **false** if the types in E are never instances of c, or the set $Z \subset E$ of types that are instances of c if the type inquiry can not be reduced to a constant.

7.3 Optimizing Dynamic Casts in Java

Optimization of dynamic casts is dependent upon the object model being used. We will begin with the simplest object model, namely Sun's Java object model in their JDK version 1 (hereafter called the "Sun object model").

optimizeInstanceOf($E \subseteq C_L, c \in C$)
    $Z \leftarrow \{z \in E : \ c \in Bases^*(z)\}$
    if $Z = E$
        return true
    else if $Z = \emptyset$
        return false
    else
        return Z

Figure 7.2: Optimizing a Dynamic Type Inquiry

In the Sun object model, an object consists of class data and a pointer to a class object, which includes the method table. In the general case, method dispatch involves a dynamic lookup in the method table, just as in the dynamically-typed languages like Smalltalk and SELF. While this sometimes extracts a performance penalty, it is simple and never requires more than one word per object for the class information. By contrast, C++ object models may require multiple words per object for the class information.

The Sun object model also simplifies casting, since there are no offsets to adjust anywhere. A cast operation is simply a check: once we are certain that the object is of the appropriate type, no further modification is necessary.

Due to this simplicity, the **optimizeInstanceOf** algorithm of Figure 7.2 can also be used for casts with the Sun object model. The cast in the Java code fragment

```
foo(Base b) {
    Derived d = (Derived) b;
    ...
}
```

can be implemented as a simple pointer copy if **optimizeInstanceOf** returns **true**, or as `throw ClassCastException(b)` if **optimizeInstanceOf** returns **false**.

7.3.1 Optimizing A Common Idiom

A common idiom in Java programs is to first test if an object is an instance of a type, and then downcast to that type if the test succeeds. For example,

```
if (p instanceof L)
  L lp = (L) p;
```

optimizeDynamicCast1$(e \in C, E \subseteq C_L, c \in C)$
    $Z \leftarrow \{z \in E :\ c \in \mathit{Bases}^*(z)\}$
    if $Z = \emptyset$
        return false
    Let $t \in Z$
    if $E = Z$ **and** $\forall_{x \in Z} :\ \mathit{ThisAdjustment}(x, e, c) = \mathit{ThisAdjustment}(t, e, c)$
        return $\mathit{ThisAdjustment}(t, e, c)$
    else
        return Z

Figure 7.3: A Simple Dynamic Cast Optimization Algorithm for use with C++-style Object Models.

This idiom is easily recognizable and can be implemented with a single type inquiry operation, provided either that there are no intervening operations between the **instanceof** and the downcast operators or that flow analysis is used to ensure that the possible types of `p` do not change between the **instanceof** and the downcast operators.

7.4 Optimizing Dynamic Casts in C++

When a more complex object model is being used in which a cast may involve an adjustment to the `this` pointer, the casting optimization becomes slightly more complex. This is the case for virtually all C++ object models.

Figure 7.3 is an algorithm for optimizing dynamic casts, which extends the algorithm of Figure 7.2 that applied to the Sun object model. The static type $e \in C$ of the expression is required as an additional parameter. The set of castable classes Z is computed in the same way, and if the set Z is empty, then the value **false** is returned, indicating that the cast will always fail and can be replaced with a `throw` clause.

If the cast will always succeed (that is, if $Z = E$), then if all possible casts lead to the same offset being added to the `this` pointer, then that offset value is returned. Otherwise the set Z is returned, indicating failure.

A more thorough algorithm is shown in Figure 7.4. This algorithm returns an empty set if the cast must fail. Otherwise, it returns a set of pairs consisting of a set of classes and the offset value to add to the `this` pointer in order to cast each of them to type c. For those classes that can not be cast to type c, the cast value $\perp$ is used.

```
optimizeDynamicCast2(e ∈ C, E ⊆ C_L, c ∈ C)
    Z ← {z ∈ E : c ∈ Bases*(z)}
    if Z = ∅
        return ∅
    Q ← ∅
    if E ≠ Z
        Q ← {< E − Z, ⊥ >}
    while Z ≠ ∅
        Let t ∈ Z
        Z ← Z − {t}
        W ← {t} ∪ {x ∈ Z : ThisAdjustment(x, e, c) = ThisAdjustment(t, e, c)}
        Q ← Q ∪ {< W, ThisAdjustment(t, e, c) >}
        Z ← Z − U
    end while
    return Q
```

Figure 7.4: A More Sophisticated Dynamic Cast Optimization Algorithm

The sets returned by algorithm **optimizeDynamicCast2** are used to determine a sequence of conditionals that test the type of the object, presumably starting with the smallest set in Q. It will often be the case that the sets of Q only contain a single element, making the conditionals quite fast.

7.5 Complexity Issues

The expensive part of all of the algorithms presented so far in this chapter is the computation of the set

$$\{z \in E : \; c \in \mathit{Bases}^*(z)\}$$

from the set E of possible class types of the expression being cast or interrogated. Since the classes of E are related, there is likely to be a lot of redundant work in calculating $\mathit{Bases}^*(e)$ for the various class types, each time involving a recursive walk up the class hierarchy.

The potential solution is to calculate $\mathit{Bases}^*(c)$ for each class $c \in C$, and store it in a hash table at the class node. This would make the computation of the above set take $O(E)$ expected time, at the cost of a considerable amount of space and pre-computation.

Even in Java, in which dynamic casts are a major part of the language idiom and occur

```
                               ; the object pointer is in R1
Load  R2, 0(R1)                ; R2 is the pointer to the VFT
Load  R3, MethodOffset(R2)     ; R3 is the method pointer
Load  R4, MethodOffset+4(R2)   ; R4 is the this adjustment
Add   R1, R1, R4               ; adjust the object pointer
Call  R3                       ; call the virtual function
```

Figure 7.5: Virtual Function Dispatch using Two-Column Virtual Function Tables

quite frequently, they are still rare compared to method invocations. Therefore, from a practical standpoint, the best approach is to calculate the set Z in a straightforward, albeit inefficient, manner.

7.6 Optimizing Virtual Dispatches

In the "reference" object model of the C++ Annotated Reference Manual (ARM) [Ellis and Stroustroup 1990], virtual function tables consist of two columns: the first column contains the pointer to the function, and the second column contains an adjustment to the `this` pointer. The position of a virtual function pointer in the table is an offset determined at compile-time.

Many C++ compilers use a variant of the ARM object model with two-column virtual function tables (VFT's). Using the ARM object model, a virtual function call is implemented as shown in Figure 7.5.

A major drawback of two-column VFT's is that the vast majority of `this` pointer adjustments are redundant because multiple inheritance is relatively rare. As long as only single inheritance is used, methods introduced by derived classes are simply assigned offsets in the VFT greater than those of the methods in the base classes. Therefore, the `this` pointer adjustment is 0.

The algorithm in Figure 7.6 formalizes this description. Given a virtual function call of the form `(expr)->foo()`, $e \in C$ is the static type of `expr`, and the set $E \subseteq C$ is the set of possible dynamic types of `expr` based on static analysis. The method m is the method name of `foo`.

The set Z is all of the different `this` pointer adjustments for method m for the classes in E. If there is only one adjustment, optimization is successful and the adjustment a is

optimizeTwoColumnDispatch$(e \in C, E \subseteq C_L, m \in M)$
  $Z \leftarrow \{z \in E: \ \mathit{ThisAdjustment}(z, e, \mathit{ClassOf}(m))\}$
  if $|Z| = 1$
    Let $a \in Z$
    return a
  else
    return $\perp$

Figure 7.6: Calculating the set of Possible This Pointer Adjustments

```
                                ; the object pointer is in R1
Load  R2, 0(R1)                 ; R2 is the pointer to the VFT
Load  R3, MethodOffset(R2)      ; R3 is the method pointer
Call  R3                        ; call the virtual function
```

Figure 7.7: Virtual Function Dispatch after Optimization

returned (usually a will be 0). If multiple offsets are possible, the value $\perp$ indicates failure and the dispatch will not be optimized.

Figure 7.7 shows the code generated when the `this` pointer offset is determined to be zero by **optimizeTwoColumnDispatch**. In the rare case when there is a constant non-zero offset x, an add immediate instruction of the form `AddI R1, R1,` x is inserted somewhere before the call.

7.6.1 Converting Java Interface Calls to Virtual Calls

The optimization we have just described for C++ has an exact analogue in Java: the conversion of interface calls to virtual calls. In Java, virtual calls are made through an implementation class, and since there is only single inheritance of implementation, the virtual function can always be obtained at a fixed offset. However, for interface calls, multiple inheritance is possible and it may be necessary to search for the target function. (Caching is used to eliminate most of these searches).

A variant of **optimizeTwoColumnDispatch** can be used to determine when an interface call can be converted into a virtual call: instead of computing the set of `this` pointer adjustments, compute the set of virtual function table offsets of the target function. If there is only one, the call can be converted into a virtual call, which is less expensive than an indirect call. Serrano [1997] reports that most interface calls in the benchmarks he examined were due to use of the `Enumeration` interface, and that almost 100% of interface calls could be converted to virtual calls using this method.

7.7 Eliminating Java Synchronization

Preliminary results from benchmarking Java applications show that they spend an enormous amount of time performing synchronization operations. Why is this? Because all of the Java library classes, in order to be thread-safe, implement their `public` methods as `synchronized` methods. Most Java programs make significant use of the library classes, and as a result call a large number of synchronized methods.

However, many Java programs, especially those designed as stand-alone applications (like `javac`, the Sun Java compiler), are single threaded. Rapid Type Analysis can be used to determine when the synchronization operations may be safely omitted by the compiler. In order for a program to be multi-threaded, it must instantiate `Thread` or one of its derived classes. Therefore, if

$$\mathit{Derived}^*(\texttt{Thread}) \cap C_L = \emptyset$$

then the program does not create any threads and the synchronization operations can be omitted.

Such optimizations, combined with a more efficient run-time implementation of locking, may well be able to reduce synchronization overhead in Java programs to the point where it is not a significant performance issue. We have recently shown that highly efficient run-time implementations of locking are possible for Java [Bacon et al. 1997].

7.8 Future Applications

We have only described some of the most obvious applications of Rapid Type Analysis in this chapter.

Sweeney and Tip [1997] have used RTA as the basis for an algorithm that finds and removes unused data members in C++ programs. The dynamic space reduction can be significant. This is yet another example of how Rapid Type Analysis and related algorithms can be used to remove excess generality from programs.

The combination of dead code removal achieved by RTA and the above dead data member removal could be combined into a very powerful application extracter. The extracter would be particularly useful in generating minimal size executable code images for embedded devices with limited memories.

In general, constructing an accurate call graph can be used to improve almost any inter-procedural optimization. At the very least, it reduces the time required to analyze the program.

Chapter 8

Optimizing Incomplete Programs

Most system code is supplied in libraries. Since our techniques all derive their power from analyzing the whole program at once, how can they be applied to libraries? Libraries actually raise two issues: the optimization of the library, given that the final class hierarchy and the final program is unknown; and the optimization of the client program, given that the source code for the library is unavailable. To a large degree these problems are symmetrical because they both involve optimization in the presence of an incomplete CHG and an incomplete PVG. However, there are some important differences.

To simplify matters, when a program is the client of multiple libraries, they can be treated as a single library for the purposes of this chapter.

In order to optimize an incomplete program, we must know which classes are being exported by the library, and which classes are purely internal. Java makes this information explicit: a class declared `public` can be instantiated outside of the package in which it is declared; a class declared as `public` and not declared as `final` can be subclassed outside of the package in which it is declared.

The set of classes that can be instantiated by the clients of the library are called the *exported* classes, denoted C_Ψ, where $C_\Psi \subseteq C$. For Java, C_Ψ contains the `public` classes. For C++, C_Ψ can be specified by pragmas in the program or in an auxiliary file, or C_Ψ can be inferred by inspecting the header files that will be delivered with the library: any class defined in the header files is placed in C_Ψ.

The classes that can be subclassed by the clients of the library are called the *derivable* classes, denoted C_Δ, where $C_\Delta \subseteq C_\Psi \subseteq C$. For Java, C_Δ contains the `public`, non-`final` classes. For C++, C_Δ can be specified by pragmas, or it can conservatively be assumed

rapidTypeAnalysis(F, S, I, R)
    $Q_V \leftarrow \emptyset$
    $C_L \leftarrow C_\Psi$
    $F_L \leftarrow S_L \leftarrow I_L \leftarrow \emptyset$
    $R \leftarrow R \cup M_\Delta \cup M_\Psi \cup F_\Psi$
    for each $f \in R$
        analyze$(f,$ **false**$)$

Figure 8.1: Rapid Type Analysis algorithm modified for use with libraries.

to be identical to C_Ψ.

In addition to information about exported classes, we must also know which methods and functions are being exported. We define M_Ψ to be the set of `public` methods of the classes in C_Ψ, M_Δ to be the set of `protected` methods of classes in C_Δ, and F_Ψ to be the set of non-methods exported by the library.

8.1 Compiling a Library

The general approach to the incomplete code problem that we take here is to first build the CHG and PVG for the incomplete program, and run the Rapid Type Analysis algorithm on the incomplete PVG. We then present modified versions of the optimizations that take the missing code into account.

8.1.1 Analyzing a Library

When analyzing a library, the unknown client program raises three issues that we must take into account. First, any classes in C_Ψ may be instantiated by the client program. Second, any accessible methods or functions in the library may be called by the client program. Third, any classes in C_Δ may be subclassed by the client program.

We use the CHG and PVG construction algorithms unchanged. The missing portions will be dealt with in the subsequent analysis and optimization.

The RTA algorithm will account for the first issue, instantiation of library classes by the client, by initializing the set of live classes C_L to C_Ψ instead of to the empty set as shown in Figure 8.1 (the original algorithm is shown in Figure 4.3 on page 47).

The second issue, calls from the client program into the library, can also be addressed in the RTA algorithm by expanding the set of roots of the call graph R to include M_Ψ (the set of `public` methods of the exported classes in C_Ψ), M_Δ (the set of `protected` methods of the derivable classes in C_Δ), and F_Ψ (the set of exported non-methods in F).

With these two simple modifications, we have handled all the potential effects of the unavailable code except for subclassing of classes in C_Δ. Subclassing will be addressed by modifying the optimization algorithms.

8.1.2 Analyzing a Library with Function Pointers

In languages like C++ that include function pointers, some changes to the extended RTA algorithm of Figures 4.4 and 4.5 are necessary in addition to the changes we have already described.

The modifications to the extended RTA algorithm are shown in Figure 8.2. To begin with, we must assume that any exported functions may have their addresses taken in the unavailable code (lines 5 and 6).

In addition, we must be more conservative in our treatment of function pointers, because once a function's address is taken, it could be passed into the user code and called from there. Therefore, we assume that any function whose address is taken in live code is itself live (line 13).

With pointer-to-member calls we do not need to be quite so conservative, because the class hierarchy gives us some additional leverage. In particular, a member function pointer can only be used with its defining class and its transitive subclasses. If none of these classes is exported, then it is not possible to invoke the member pointer from the unavailable code (lines 20 and 21).

8.1.3 Virtual Function Resolution

So far we have described how to analyze an incomplete program, and have taken into account all effects of the unavailable code except for derivation from classes in C_Δ. The modified indirect function call resolution algorithm is shown in Figure 8.3 (for comparison, the original algorithm is in Figure 5.1(a) on page 59).

To begin with, the new algorithm makes no attempt to resolve the function pointer calls in S_P because another function of the correct type could be in the unavailable code and have its address passed into the library code (line 3).

```
extendedRTA(F, S, I, R)
    Q_V ← Q_P ← Q_M ← ∅
    C_L ← C_Ψ
    F_L ← S_L ← I_L ← ∅
    F_A ← F_Ψ
    M_A ← M_Δ ∪ M_Ψ
    R ← R ∪ M_Δ ∪ M_Ψ ∪ F_Ψ
    for each f ∈ R
        analyze(f, false)

addFunctionPointers(f ∈ F)
    for each g ∈ FunctionPointers(f) :  g ∉ F_A
        F_A ← F_A ∪ {g}
        analyze(g, false)
        for each i ∈ I : < g, i >∈ Q_P
            Q_P ← Q_P − {< g, i >}
            addCall(i)

addMemberPointers(f ∈ F)
    for each m ∈ MemberPointers(f) :  m ∉ M_A
        M_A ← M_A ∪ {m}
        if Derived*(ClassOf(m)) ∩ C_Ψ ≠ ∅
            analyze(m, false)
        for each i ∈ I : < m, i >∈ Q_M
            Q_M ← Q_M − {< m, i >}
            Let < s, g, h, P >= i
            if P =⊥ or P ∩ C_L ≠ ∅
                addCall(i)
            else
                addVirtualMappings(P, i)
```

Figure 8.2: Modifications to the extended RTA algorithm of Figures 4.4 and 4.5 to handle function pointers in the presence of libraries.

1 **resolveCalls**(F, S_L, I_L, C_Δ)
2 $\quad S_R \leftarrow \emptyset$
3 $\quad$ **for each** $s \in S_L : \; s \in S_V$ **or** $s \in S_M$
4 $\quad\quad Q \leftarrow \{< s, f, g, P > \in I \mid f \in F, g \in F, P \in 2^C\}$
5 $\quad\quad$ **if** $\forall_{<s,f,g,P> \in Q} : \; (P = \perp$ **or** $P \cap C_\Delta = \emptyset)$
6 $\quad\quad\quad R \leftarrow \{i \in Q : \; i \in I_L\}$
7 $\quad\quad\quad$ **if** $|R| = 1$
8 $\quad\quad\quad\quad S_R = S_R \cup \{s\}$

Figure 8.3: Indirect Function Call Resolution for Libraries

For each virtual or pointer-to-member call site s, the set Q of call instances is computed (line 4). We then check to make sure that all of the call instances in Q are either non-virtual member-pointer calls ($P = \perp$), or if they are virtual calls that none of the possible classes is derivable ($P \cap C_\Delta = \emptyset$). If this check fails, then we do not attempt to resolve the call.

How does this work? The only way for the client code to "invalidate" a virtual call resolution is by subclassing one of the possible classes at the call site, overriding the method in question, and then passing an object of the derived type into the client code. If none of the possible classes (in the sets P) can be subclassed, then it will not be possible for the client code to invalidate the virtual call resolution).

If the check on line 5 succeeds, then if there is only one live call instance in Q, the call is resolved (lines 6–8).

8.1.4 De-virtualizing Inheritance

Figure 8.4 shows the modifications to the algorithm for de-virtualizing inheritance. Lines 1–11 are unchanged. Lines 12–15 take into account subclassing of classes in C_Δ by the unavailable code.

There are two situations in which a virtual inheritance edge in the library can not be de-virtualized, even if the library does not multiply inherit the base class. The first, shown in Figure 8.5, occurs when both the base class J and the derived class K of the virtual inheritance edge are in C_Δ. In this case, the unavailable code may also virtually derive class L from K, and then derive class M from both the library class K and its own class L. Since there are two virtual paths from M to J, the virtual edges can not be de-virtualized.

devirtualizeBases$(C, D, V, C_\Omega, C_\Delta)$
    for each $d \in D$
        if *IsVirtual*(d)
            mustBeVirtual(d) = **false**
    for each $c \in C_\Omega$
        $Q \leftarrow \emptyset$
        initializeCounts(c)
        $Q \leftarrow \emptyset$
        findVirtualBases(c)
        $Q \leftarrow \emptyset$
        markVirtualEdges(c)
    for each $c \in C$
        for each $d \in C_\Delta : < c, d > \in D$ **and** *IsVirtual*$(< c, d >)$
            if $c \in C_\Delta$ **or** $(\exists_{e \in C_\Delta} : < c, e > \in D$ **and** *IsVirtual*$(< c, e >))$
                mustBeVirtual$(< c, d >) \leftarrow$ **true**

Figure 8.4: Modifications to the algorithm for de-virtualizing inheritance.

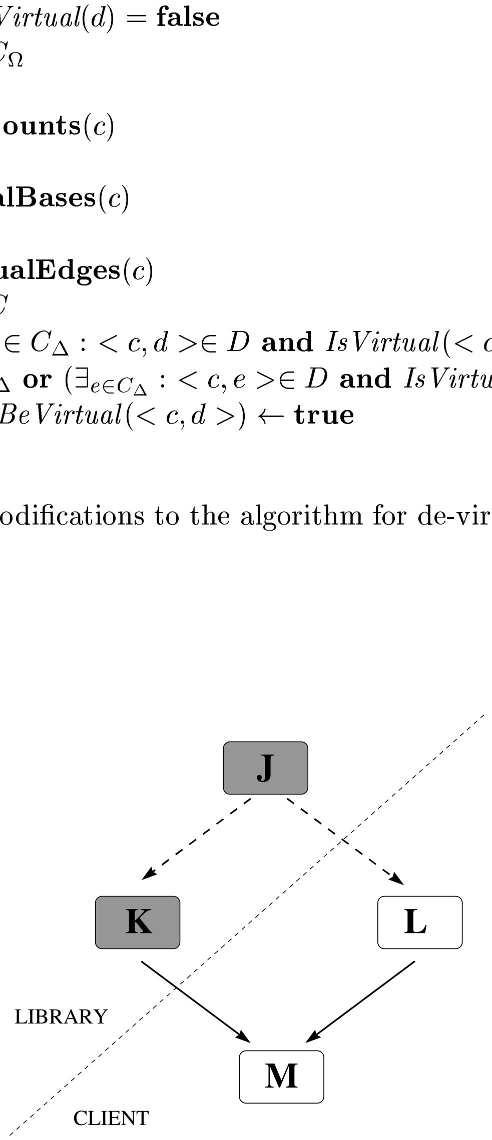


Figure 8.5: Inability to De-virtualize Inheritance in a Library: Case 1. Shaded classes J and K are subclassable library classes (they are in C_Δ). Dashed inheritance arcs are virtual. Inheritance arc J-K must be virtual because the client program could introduce classes L and M.

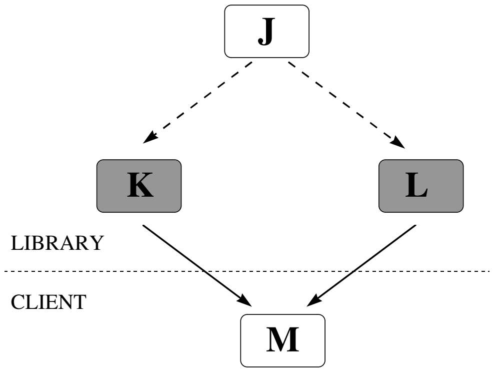


Figure 8.6: Inability to De-virtualize Inheritance in a Library: Case 2. Shaded classes K and L are subclassable library classes (they are in C_Δ). Dashed inheritance arcs are virtual. Inheritance arcs J-K and K-L must be virtual because the client program could introduce class M.

This case is taken into account in the first part of the conditional clause on line 14 of the new de-virtualizaion algorithm: if the class c is in C_Δ as well as d, then the edge $< c, d >$ must be virtual.

The second case is shown in Figure 8.6: the inheritance hierarchy is the same, but this time J, K, and L are library classes but only K and L are in C_Δ. The client code could derive class M from K and L, in which case neither of the edges could be de-virtualized. This case is taken into account in the second part of the conditional on line 14 of the new de-virtualization algorithm: if $< c, d >$ is virtual and $d \in C_\Delta$, then $< c, d >$ must be virtual if there is another virtual edge $< c, e >$ such that $e \in C_\Delta$.

8.1.5 Type Inquiries and Type Casts

All of the optimizations of Chapter 7 involve an expression whose static type is $e \in C$, for which a set of possible types E is provided by RTA or some other analysis. In the case of RTA, the set of possible types is

$$E = \mathit{Derived}^*(e) \cap C_L.$$

All of these optimizations rely in some way on the set of possible types being determined at compile-time. If this set can not be determined statically, then the optimizations can not be performed.

By placing all of the library classes that might be instantiated by the client program C_Ψ into the set of live classes C_L, we have accounted for expansion of the set E by known classes. However, we must also account for expansion of the set E by unknown classes — classes that will be defined in the client code.

Accounting for unknown classes is actually quite simple. An unknown class would cause the set $\mathit{Derived}^*(e)$ to grow, by being derived from one of the classes in $\mathit{Derived}^*(e)$. But we know exactly which classes can be subclassed by the client program: the classes in C_Δ. Therefore, any of the type inquiry or type casting optimizations of Chapter 7 can be adapted for libraries simply by checking if

$$\mathit{Derived}^*(e) \cap C_\Delta \neq \emptyset$$

in which case the optimization can not be applied. Otherwise, there is no possibility that the set E will be expanded by the client program, and the optimization is safe.

8.2 Optimizing Library Clients

We now turn to the problem of optimizing a client program in the absence of source code for one or more libraries it is using. This assumes some sort of separate compilation model, which at this point Java does not have: to compile a program all the bytecodes that it makes use of must be available. From our perspective, bytecodes are "source-level" information because they contain all of the information necessary to type-check the program.

Because of the absence of a separate compilation model for Java, our description in this section will be for C++. However, it is straightforwardly adaptable to Modula-3, Dylan, or a foreseeable separate compilation scheme adopted for Java.

In order to type-check a program that uses (or subclasses) a library class, the public (or public and protected) member declarations of the library class and all of its base classes must be available. For C++, this means that the class declarations of all classes in C_Ψ must be available in the library header files (that is, all source code associated with the classes in C_Ψ except for the method bodies). Therefore, we know that the complete class hierarchy with respect to the classes defined in or used by the client code is available.

As with libraries, we will use the unmodified CHG and PVG construction algorithms to construct the partial CHG and partial PVG of relevance. In this case, the PVG may contain call instances that refer to methods for which we have no source information; we will simply treat them as leaf procedures.

Optimizing a library client in the absence of the library is remarkably similar to optimizing a library in the absence of the client, because in both cases we must account for the effect of the unavailable code on the library classes. In the case of a client program, we know about all (or almost all, as will be explained below) of the operations upon the classes introduced by the client — since they are introduced by the client, they can not be referred to by the library.

What we do not know is what library classes are instantiated in the library, and what library classes are subclassed in the library. Therefore, the definition of C_Ψ is unchanged: it is the set of classes exported by the library. However, the corresponding set M_Ψ of methods that could be invoked or have their addresses taken in unavailable code is the set of *all* methods of the classes in C_Ψ.

In the absence of any pragmas, we must assume that C_Δ, the set of classes that may be subclassed in unavailable code, is the same as the set C_Ψ. In languages that support `final` classes in the style of Java (such as Dylan [Feinberg et al. 1997]), C_Δ is the set of non-`final` classes in C_Ψ.

The set F_Ψ of exported functions is the set of all function (non-method) interfaces available to the library client.

8.2.1 Changes to the RTA Algorithm

The RTA algorithm of Figure 8.2 can be used, although some additional modifications are required. In particular, we will not be adding elements to the set R of roots of the PVG, because the roots have been supplied as part of the input.

In addition to the altered handling of function and member-function pointers, there is another issue that must be addressed. If the client code subclasses a library class and overrides one of its functions, there may be a virtual dispatch in the library code that could invoke the overriding function in the client code. Therefore, as soon as a class from the client code is instantiated, all of its methods that override methods in the library code must be assumed to be live.

The corresponding modifications to the RTA algorithm are shown in Figure 8.7. If a non-library class is instantiated (line 9), then for any non-virtual methods that are not already part of the call graph (line 10), we check whether they might be called from within

instantiate($c \in C$)
    if $c \in C_L$
        return
    $C_L \leftarrow C_L \cup \{c\}$
    for each $i \in I : < c, i > \in Q_V$
        if $i \notin I_L$
            addCall(i,**false**)
            $Q_V \leftarrow Q_V - \{< c, i >\}$
    if $c \notin C_\Psi$ **and** $Bases^*(c) \cap C_\Delta \neq \emptyset$
        for each $m \in M, d \in C : < c, m, d > \in V$ **and** $IsVirtual(m)$ **and** $m \notin F_L$
            $Q \leftarrow \emptyset$
            if overridesLibraryMethod($< c, m, d >$)
                analyze(m, **false**)

overridesLibraryMethod($v \in V$)
    Let $< c, m, d >= v$
    if $d \in C_\Psi$
        return true
    if $c \in Q$
        return false
    $Q \leftarrow Q \cup \{c\}$
    for each $b \in C : < b, c > \in D$
        if $\exists_{n \in M, e \in C} : < b, n, e > \in V$ **and** $Sig(m) = Sig(n)$
            if overridesLibraryMethod($< b, n, e >$)
                return true
    return false

Figure 8.7: When a non-library class is instantiated, if any of its methods override methods of library classes, they must be assumed to be live.

the library code (lines 11–13). The set Q keeps track of which classes have been visited by the recursive **overridesLibraryMethod** function, to ensure that no class and its bases is processed more than once.

The **overridesLibraryMethod** function walks up the class hierarchy from the class c, looking for a visible method defined by one of the library classes in C_Ψ. As we have mentioned previously, backpointers are kept from visible methods to the corresponding visible methods in the base classes, so the iteration on lines 21 and 22, while appearing complex, is implemented as an iteration over a list.

These modifications to the RTA algorithm do impact its complexity. A single invocation of **overridesLibraryMethod** could take $O(D \log C)$ time (the function is a "depth-first" traversal up the class hierarchy graph). Each class in $C - C_\Psi$ can be instantiated at most once, and has at most $\mathcal{M}$ virtual methods. Therefore, the contribution of the library modifications is

$$O((C - C_\Psi)\mathcal{M}D \log C)$$

and the total worst-case complexity of RTA is degraded to

$$O(R + I\mathcal{P} \log C + (C - C_\Psi)\mathcal{M}D \log C).$$

However, since class hierarchies are almost always quite shallow, the cost of **overridesLibraryMethod** will normally be bounded by a small constant, and the $D \log C$ term can be dropped. The expected complexity is therefore

$$O(I\mathcal{P} + (C - C_\Psi)\mathcal{M})$$

and by noting that only live classes are instantiated and therefore the term for the number of classes checked can be sharpened to $C_L - C_\Psi$, which is necessarily smaller than I, we can use the slightly worse but more intuitive bound

$$O(I(\mathcal{P} + \mathcal{M})).$$

8.3 Traditional Separate Compilation

In a traditional file-at-a-time compilation environment, can the optimizations presented in this dissertation be applied? There are two possibilities.

First, each file can be compiled as though it were a stand-alone library, using the algorithms of Section 8.1. The set of "exported" classes C_Ψ is the set of all classes not defined locally in the file being compiled. The set of classes C_Δ must be the same as the set of classes C_Ψ.

Unfortunately, unless a lot of classes are file-local (which in our experience is unusual), this approach is not likely to be very effective.

A much better approach is to *gather* the necessary information at compile-time and place it in the object files, and then *optimize* the program at link-time when the entire program is available.

This approach allows the full power of the RTA algorithm to be applied, but restricts the power of the optimizations. In particular, while it is possible to resolve virtual and indirect calls, it is no longer possible to inline them. And while optimization of type inquiries and type casts is possible, de-virtualization of inheritance (which relies on changing the global class structure of the program) is not.

Chapter 9

Conclusion

We have shown how a fast, simple algorithm — Rapid Type Analysis — can be used to drive a suite of optimizations of the most expensive features of object-oriented programming languages. These optimizations allow the programmer to design a system extensibly, by using virtual functions and virtual base classes, without having to pay for that flexibility unless it is really needed. We hope and believe that Rapid Type Analysis can help raise the level at which programmers use languages like C++ and Java.

Rapid Type Analysis suffers from only one limitation: it requires access to the whole program (or library) in order to optimize it effectively. While this is at odds with traditional separate compilation, current trends are towards integrated development environments in which the compiler has a more global view of the program. In addition, the emergence of Java bytecodes [Lindholm and Yellin 1997] as a high-level architecture-neutral distribution format means that more and more programs will be available to the compiler in their entirety.

Rapid Type Analysis has many advantages:

- It is very fast: the median compile-time overhead for analysis that we measured for real C++ programs was 1.4%, with a maximum of 5.6%. This makes RTA practical for everyday use.
- It is effective: in five out of seven large C++ benchmarks, it resolved all or almost all of the monomorphic virtual calls.
- It reduces code size: by an average of 25% in the benchmarks we measured.

- It can *speed up* compilation: by removing the need to perform the back-end phases of compilation on the eliminated functions. Grove et al [1997] have implemented RTA in the Vortex compiler and found that RTA sped up compilation of 6 out of 12 Cecil and Java benchmarks.

- It is scalable: the expected complexity is proportional to I, the set of call instance edges. This has been confirmed experimentally, both in our measurements and in those of Grove et al.

While there is still debate on how much additional precision can be achieved in practice by algorithms that compute per-variable type information, there is no question that RTA is the best algorithm that is practical for everyday use.

RTA is currently being incorporated into two products by IBM: a C++ compiler and a Java compiler. RTA has also been implemented by Craig Chambers' group at the University of Washington. RTA has become the new standard benchmark of performance in both time and precision for type analysis algorithms.

9.1 Problems with C++

In the process of doing our dissertation research, we spent several years studying the C++ language, developing compiler optimizations for C++, implementing them in a C++ compiler, and working with C++ benchmark programs. I regret to say that the experience has not been a pleasant one, and we would caution any researchers about to embark on a compiler project involving C++ to reconsider.

C++ is such a complex language that just building a front-end for the language, typically the easiest part of a compiler project, is an enormous undertaking. Our work was repeatedly delayed because the compiler that we were depending upon had not yet implemented parsing or checking of some obscure language feature.

Furthermore, these complexities are not inherent in object-oriented languages. Our more recent research with Java has been notably simpler and less painful. While C++ has served the valuable function of bringing object-oriented programming into the mainstream, it lacks the orthogonality and simplicity of a language that can stand the test of time.

The complexities of C++ filter down through all levels of the compiler and run-time system, adding significant amounts of complexity at every layer. The end result is that working with C++ is considerably harder than working with other languages. In this dissertation, the evidence of the complexity can be seen in the numerous sections adapting the algorithms to the additional features of C++: two types of method calls, two types of inheritance, pointer-to-member functions, construction virtual function tables, and so on.

Let the researcher beware.

9.2 Future Work

9.2.1 Fast Type Analysis

Are there effective algorithms in the space between RTA and the fastest algorithms that compute per-variable information, like Steensgard's [1996a]? We think so.

In languages like Java, the absence of support for parametric polymorphism in the language forces the use of a programming idiom in which objects of type `Object` are returned from a container or iterator class. Algorithms that compute per-variable information are likely have more of an advantage in precision over RTA in such languages than in C++. However, we expect that extending RTA with some simple heuristics will go a long way to narrowing that gap.

In particular, Serrano [1997] has implemented RTA in a Java compiler. He observed that a large proportion of unresolved calls were to the `next()` method of the `Enumeration` class. He is extending RTA by making use of return type information in a simple local (per-method) analysis which will be able to resolve these calls and others like them.

Another area that we believe could be fruitful is simple cloning for private objects. It should be possible to determine quickly whether a private object can ever be accessed from outside its containing object. If it can not be accessed non-locally, and if the set of types within the object can be constrained, then the private object could be customized to the local types.

For example, if an object contains a private object of class `Stack`, and if the stack is not accessible outside of the object, then we may be able to quickly determine that the stack only contains objects of type `Foo`, and create a customized `FooStack`.

9.2.2 Fast Run-time Implementation

The other side of the coin, which we have not had the scope to address in this dissertation, is what to do when the expensive operations can not be eliminated at compile-time.

This is just as important in helping to make the high-level features of a language usable in practice.

While a great deal of effort has been expended on fast method dispatch techniques, much less has been done to reduce the cost in time and space of multiple (virtual) inheritance, or to reduce the cost of dynamic casting, which is very important in Java.

Most glaring right now is the overhead of `synchronized` methods in Java, where single-threaded programs may spend 50% of their time in synchronization overhead.

The key to all of these run-time optimizations is always the same: make the common case fast. To this end, we need measurements of the behavior of multiple inheritance, dynamic casting, and synchronization in real object-oriented programs.

Appendix A

Notation

This appendix describes notational conventions and defines all of the variables (except temporaries), predicates, and functions used in this dissertation.

The following conventions are used with regard to variable names:

- Caligraphic letters ($\mathcal{B}$, $\mathcal{M}$) represent numeric quantities used in complexity calculations;
- Lower-case letters (c, m) represent scalars or tuples;
- Upper-case letters (C, F) represent sets;
- Subscripted upper-case letters denote subsets (C_L is a subset of C);
- Q, W, and Z denote temporary sets.

Iteration over sets of values is denoted by the **for each** operator to avoid confusion with the quantifier $\forall$.

The set of all possible subsets of a set X (the power set of X) is denoted 2^X.

Set constructors have a three-part general form consisting of the elements, the free variables, and the conditions. For example,

$$\{< s, f, g, P >\in I \mid f \in F, g \in F, P \in 2^C :\ P \cap C_L \neq \emptyset\}.$$

However, when it is unambiguous the free variable can be combined with the element expression as in

$$\{b \in C :< b, c >\in D\}$$

and when there is no condition other than set membership, the condition clause may be omitted as in

$$\{\mathit{TypeOf}(f) \mid f \in F\}.$$

A.1 Symbol Glossary

$\mathcal{B} = \max_{c \in C} |Bases(c)|$ is the maximum number of base classes of any class (in other words, the maximum in-degree of the class hierarchy).

$Bases(c) = \{b \in C :< b, c >\in D\}$, where $c \in C$, is the set of base classes of c.

$Bases^*(c)$, where $c \in C$, is the set of all transitive base classes of c, including c itself.

C is the set of class nodes (page 17).

$C_L \subseteq C$ is the set of "live" classes, as determined by static analysis. Since C_L is necessarily conservative, this means that if $c \in C - C_L$, then objects of type c are guaranteed not to be created during any execution of the program.

$C_\Delta \subseteq C$ is the set of classes in the library that can be subclassed.

$C_\Psi \subseteq C$ is the set of classes in the library that can be instantiated.

$C_\Lambda \subseteq C$ is the set of classes in the reduced CHG determined by C_L. This includes all classes in C_L and their (transitive) base classes.

$C_\Omega \subseteq C_L$ is the set of leaf classes of C_Λ.

CHG $=< C, D, V >$ is the class hierarchy graph.

$ClassOf(m)$, where $m \in M$, is the class $c \in C$ in which the method m is defined.

D is the set of derivation edges $< c, d >$ where $c, d \in D$ (page 17). Derivation edges point from the base class to the derived class (c is the base class, d is the derived class).

$Derived(c) = \{d \in C : < c, d >\in D\}$, where $c \in C$, is the set of all classes derived from c.

$Derived^*(c)$, where $c \in C$, is the set of all classes derived from c, including c itself (page 19).

F is the set of all functions defined by the program, both methods and non-methods. F does not include interfaces, only functions and methods that have code bodies. Therefore, $M_D \subseteq F$. (Page 34)

$F_A \subseteq F$ is the set of functions whose addresses have been taken in the code bodies of the functions in F_L.

$F_L \subseteq F$ is the set of "live" functions.

$F_\Psi \subseteq F$ is the set of exported functions in the library.

$FunctionPointers(f) \subseteq F$, where $f \in F$, is the set of functions whose addresses are taken in the code body of f.

I is the set of call instances of the program (page 34). There is one call instance for each possible target at each call site.

$I_L \subseteq I$ is the set of "live" call instances.

Inherit(v), where $v =< c, m, d >\in V$, is the set of classes in *Derived*$^*(c)$ that inherit method m (page 25).

IsConstructor(f), where $f \in F$, is a boolean predicate that determines whether f is a constructor method or not.

IsBaseConstructorCall(s), where $s \in S$, is a boolean predicate that determines whether s is a call site that invokes a base class constructor as part of the process of constructing a derived object, or some other call.

IsVirtual(d), where $d \in D$, si a boolean predicate that is true if the inheritance edge d is a virtual inheritance edge (in the C++ sense).

$\mathcal{M} = \max_{c\in C} |V_c|$ is the maximum number of visible methods at any class in C.

$M = M_D \cup M_I$ is the set of all methods, both interfaces and concrete methods (page 19).

$M_A \subseteq M_D$ is the set of methods whose addresses have been taken as member pointers in the code bodies of the functions in F_L.

$M_D \subseteq M$ is the set of methods (with code bodies) defined by the program (page 19).

$M_I \subseteq M$ is the set of method interfaces introduced in a program (page 19).

$M_\Delta \subseteq M$ is the set of `protected` methods of the classes in C_Δ.

$M_\Psi \subseteq M$ is the set of `public` methods of the classes in C_Ψ.

MemberPointers$(f) \subseteq M_D$, where $f \in F$, is the set of methods whose addresses have been taken as member pointers in the code body of f.

Override(v), where $v =< c, m, d >\in V$, is the set of visible methods of *Derived*$^*(c)$ that directly override m (page 25).

$\mathcal{P} = \max_{<s,f,t,P>\in I} |P|$ is the maximum number of possible classes associated with a virtual call target (page 50). Since the sets P are taken from the *Inherit* sets, $\mathcal{P}$ is bounded above by $\max_{v\in V} |\mathit{Inherit}(v)|$.

Private(m), where $m \in M$, is a boolean predicate that is true if the method m has been declared to be `private`, meaning that it is not inherited by its subclasses.

PVG $=< F, S, I, R >$ is the program virtual-call graph.

$R \subseteq F$ is the set of root functions of the call graph, generally the `main()` function and the constructors for any global scope objects.

S is the set of call sites in the program (page 34).

$S_D \subseteq S$ is the set of direct call sites in the program.

$S_M \subseteq S$ is the set of pointer-to-member call sites in the program.

$S_P \subseteq S$ is the set of function pointer call sites in the program.

$S_V \subseteq S$ is the set of virtual call sites in the program.

$S_L \subseteq S$ is the set of "live" call sites.

$S_R \subseteq S$ is the set of call sites that have been resolved to only have one target.

Σ is the set of tuples representing the source-level call site information.

$\Sigma_D \subseteq \Sigma$ is the set of tuples $< s, f, t >$, where $s \in S$, $f \in F$, $t \in F$ that represent direct call sites in the source program.

$\Sigma_M \subseteq \Sigma$ is the set of tuples $< s, f, c, b, t >$, where $s \in S$, $f \in F$, $c \in C$, $b \in C$, $t \in T$, that represent pointer-to-member function call sites in the source program.

$\Sigma_P \subseteq \Sigma$ is the set of tuples $< s, f, t >$, where $s \in S$, $f \in F$, $t \in T$ that represent function pointer call sites in the source program.

$\Sigma_V \subseteq \Sigma$ is the set of tuples $< s, f, v >$, where $s \in S$, $f \in F$, $v \in V$ that represent virtual call sites in the source program.

$\mathit{Sig}(m)$, where $m \in M$, is the signature of method m. The signature of a method consists of its name and its type. The type of a method consists of its return type and its parameter types (see *TypeOf*, below).

$T = \{\mathit{TypeOf}(f) \mid f \in F\}$ is the set of function types.

$\mathit{ThisAdjustment}(c, f, t)$, where $c, f, t \in C$, is the adjustment required to cast the `this` pointer of an object whose actual class is c from class f to class t.

$\mathit{ThisEscapes}(m)$, where $m \in M_D$, is a boolean predicate that determines whether method m could allow its `this` pointer to be used for a virtual function call.

$\mathit{TopNum}(c)$, where $c \in C \cup \{\perp\}$, is the topological number of class c. This number has the property that if $d \in \mathit{closure}(c)$ and $d \neq c$, then $\mathit{TopNum}(c) < \mathit{TopNum}(d)$. If $c = \perp$, $\mathit{TopNum}(c) = 0$ (page 21).

$\mathit{TypeOf}(f)$, where $f \in F$, is the type of a function or method.

V is the set of visible methods (page 17).

V_c, where $c \in C$, is the subset of visible methods V of class c. Formally, $V_c = \{< c, m, d > \in V \mid m \in M, d \in D\}$.

Appendix B

Measurement Data

We have attempted to present information graphically whenever possible to allow the reader to obtain a more intuitive understanding of the results. However, some readers, in particular those attempting to reproduce our results, will find the actual numeric data much more useful. It is therefore presented in its entirety in this appendix, with cross-references to the appropriate figures.

Note that the data is presented slightly differently and sometimes with more detail here than in the graphs, which were sometimes slightly simplified to aid interpretation. In the graphs, each bar shows total quantities. In the data below, the differential contributions of each type of analysis are shown rather than the totals (remember that CHA is strictly more precise than UN, and RTA is strictly more precise than CHA). Therefore, the sum of each line of the tables represents the total quantity (virtual function calls, code size, etc).

Also, for static measurements the tables show two different portions of the "region of opportunity" for improvement by static analysis. For instance, Table B.3 shows both call sites that were monomorphic during the execution trace, and call sites that were not executed at all. In the graph, these two categories are combined into the "maximum possible" number of resolvable call sites.

Program	Direct Function	Indirect Function	Direct Method	Virtual Method	Ptr. to Member
sched	80	1	73	30	0
ixx	135	1	458	171	1
lcom	339	0	940	374	0
hotwire	129	0	420	1	0
simulate	24	2	99	16	0
idl	179	12	259	432	0
taldict	17	0	15	15	0
deltablue	102	7	88	4	0
richards	25	0	40	3	0

Table B.1: Static Distribution of Function Call Types (see Figure 5.3).

Program	Direct Function	Indirect Function	Direct Method	Virtual Method	Ptr. to Member
sched	110867	1	1223340	967795	0
ixx	94523	106	106731	45991	1041
lcom	1300743	0	1809999	1099317	0
hotwire	86254	1	69402	33504	0
simulate	9757	2	36932	10848	0
idl	7606	12	11997	14211	0
taldict	4876	0	4335589	35060980	0
deltablue	31443	220096	101389	205100	0
richards	272	0	1749610	657900	0

Table B.2: Dynamic Distribution of Function Call Types (see Figure 5.4).

Program	Resolved by			Unresolved		
	UN	CHA	RTA	Monomorphic	Unexecuted	Polymorphic
sched	0	10	0	23	0	0
ixx	66	230	5	16	67	15
lcom	14	111	0	42	148	131
hotwire	0	0	6	0	0	0
simulate	4	4	0	0	0	0
idl	866	184	91	22	34	1
taldict	8	4	2	0	0	0
deltablue	0	1	1	0	0	2
richards	0	0	0	0	0	1

Table B.3: Static Resolution of Virtual Call Sites (see Figure 5.6).

Program	Resolved by			Unresolved	
	UN	CHA	RTA	Monomorphic	Polymorphic
sched	0	91945	0	875844	0
ixx	3408	26785	8220	835	6743
lcom	63464	23841	0	488846	520050
hotwire	0	0	33598	0	0
simulate	0	10844	0	0	4
idl	4984	8576	156	457	36
taldict	16319300	12495100	6246600	0	0
deltablue	0	3	3	0	205097
richards	0	0	0	0	657900

Table B.4: Dynamic Resolution of Virtual Call Sites (see Figure 5.7).

Program	Eliminated by		Not Eliminated	
	CHA	RTA	Unexecuted	Live
sched	1644	0	6684	91560
ixx	33540	1216	73692	70188
lcom	13920	392	25172	124548
hotwire	14924	544	3652	26296
simulate	11016	0	2712	15172
idl	26868	16436	93016	107428
taldict	12876	44	0	7596
deltablue				
richards	0	72	304	9368

Table B.5: Affect of Analysis on Code Size (see Figure 5.8).

Program	Eliminated by		Not Eliminated	
	CHA	RTA	Unexecuted	Live
sched	45	0	49	143
ixx	359	14	395	340
lcom	157	6	120	496
hotwire	156	21	15	38
simulate	147	0	21	74
idl	154	121	328	253
taldict	377	3	0	49
deltablue	23	2	0	78
richards	4	1	2	71

Table B.6: Elimination of Dead Functions by Static Analysis (see Figure 5.9).

Program	Eliminated by		Not Eliminated	
	CHA	RTA	Unexecuted	Live
sched	2	0	33	23
ixx	150	55	1402	143
lcom	103	139	2914	505
hotwire	9	68	4	2
simulate	13	0	9	19
idl	986	655	1428	417
taldict	96	6	5	9
deltablue	4	2	0	5
richards	0	1	0	4

Table B.7: Elimination of Virtual Call Arcs by Static Analysis (see Figure 5.10).

Benchmark	RTA Dead	Not Dead/Not Used	Dynamically Live
sched	7	2	37
ixx	25	32	43
lcom	7	30	40
hotwire			
simulate	19	5	18
idl	7	57	22
taldict	45	0	10
deltablue	3	0	7
richards	0	2	10

Table B.8: Live Classes: RTA algorithm vs. dynamic trace (see Figure 5.5).

Bibliography

AGESEN, O. 1994. Constraint-based type inference and parametric polymorphism. In LE CHARLIER, B., ED., *Proceedings of the First International Static Analysis Symposium*, (Namur, Belgium, Sept.). Springer-Verlag, pp. 78–100.

AGESEN, O. 1995. The Cartesian product algorithm: simple and precise type inference of parametric polymorphism. In OLTHOFF, W., ED., *Proceedings of the Ninth European Conference on Object-Oriented Programming – ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, (Aarhus, Denmark, Aug.). Springer-Verlag, Berlin, Germany, pp. 2–26.

AGESEN, O. AND HÖLZLE, U. 1995. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Austin, Tex., Oct.). *SIGPLAN Not.*, *30*, 10, 91–107.

AGESEN, O., PALSBERG, J., AND SCHWARTZBACH, M. I. 1995. Type inference of SELF: analysis of objects with dynamic and multiple inheritance. *Softw. Pract. Exper.*, *25*, 9 (Sept.), 975–995.

AGESEN, O. AND UNGAR, D. 1994. Sifting out the gold. Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, OR, Oct.). *SIGPLAN Not.*, *29*, 10, 355–370.

AIGNER, G. AND HÖLZLE, U. 1996. Eliminating virtual function calls in C++ programs. In *Proceedings of the Tenth European Conference on Object-Oriented Programming – ECOOP'96*, volume 1098 of *Lecture Notes in Computer Science*, (Linz, Austria, July). Springer-Verlag, Berlin, Germany, pp. 142–166.

AMIEL, E., GRUBER, O., AND SIMON, E. 1994. Optimizing multi-method dispatch using compressed dispatch tables. In *Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, OR, Oct.). *SIGPLAN Not.*, *29*, 10, 244–258.

ANDRE, P. AND ROYER, J.-C. 1992. Optimizing method search with lookup caches and incremental coloring. In *Proceedings of the 1992 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Vancouver, B.C., Oct.). *SIGPLAN Not.*, *27*, 10, 110–126.

ANGUS, I. G. 1993. Applications demand class-specific optimizations: C++ compiler can do more. *Scientific Programming*, *2*, 123–131.

APPLE COMPUTER, INC. 1988. *Object Pascal User's Manual*. Cupertino, Calif.

ATKINSON, R. G. 1986. Hurricane: an optimizing compiler for Smalltalk. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, Ore., Sept.). *SIGPLAN Not.*, *21*, 11 (Nov.), 151–158.

BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, *26*, 4 (Dec.), 345–420.

BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. 1997. Thin locks: Featherweight synchronization for java. Submitted for publication, Oct. 1997.

BACON, D. F. AND SWEENEY, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (San Jose, Calif., Oct.). *SIGPLAN Not.*, *31*, 10, 324–341.

BALLARD, M. B., MAIER, D., AND WIRFS-BROCK, A. 1986. QUICKTALK: a Smalltalk-80 dialect for defining primitive methods. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, Ore., Sept.). *SIGPLAN Not.*, *21*, 11 (Nov.), 140–150.

BANK, J. A., LISKOV, B., AND MYERS, A. C. 1996. Parameterized types and Java. Tech. Rep. MIT LCS TM-553, Laboratory for Computer Science, Massachussets Institute of Technology, (May).

BARTON, J., CHEE, Y.-M., CHARLES, P., KARASICK, M., LIEBER, D., AND NACKMAN, L. 1994. CodeStore – infrastructure for C++-knowledgeable tools. In *OOPSLA'94 Workshop on Object-Oriented Compilation Techniques*, (Portland, Ore., Oct.).

BECK, K. 1994. Writing high-performance Smalltalk programs. In *Proceedings of OOP'94/C++ World*, (Munich, Germany, Jan.). *OOP '94/C++ World. Conference Proceedings*, p. 29.

BIRTWISTLE, G. M., DAHL, O.-J., MYHRHAUG, B., AND NYGAARD, K. 1973. *Simula Begin*. Auerbach, Philadelphia, Penn.

BLACK, A., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. 1987. Distribution and abstract types in Emerald. *IEEE Trans. Softw. Eng.*, *SE-13*, 1 (Jan.), 65–76.

BOBROW, D. G., DEMICHIEL, L. G., GABRIEL, R. P., KEENE, S. E., KICZALES, G., AND MOON, D. A. 1988. Common Lisp Object System specification X3J13 document 88-002R. *SIGPLAN Not.*, *23*, special issue (Sept.), 1–48.

BOEHM, H. J. 1989. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Portland, Ore., June). *SIGPLAN Not.*, *24*, 7 (July), 192–206.

BOTHNER, P. 1997. A gcc-based Java implementation. In *Digest of Papers, Spring COMPCON 1997, Forty-Second IEEE Computer Society International Conference*, (San Jose, Calif., Feb.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 174–178.

BRUCE, K. B., SCHUETT, A., AND VAN GENT, R. 1995. PolyTOIL: a type-safe polymorphic object-oriented language. In OLTHOFF, W., ED., *Proceedings of the Ninth European Conference on Object-Oriented Programming – ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, (Aarhus, Denmark, Aug.). Springer-Verlag, Berlin, Germany, pp. 27–51.

BURKE, M., CARINI, P., CHOI, J.-D., AND HIND, M. 1994. Flow-insensitive interprocedural alias analysis in the presence of pointers. In PINGALI, K., BANERJEE, U., GELERNTER, D., NICOLAU, A., AND PADUA, D., EDS., *Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, (Ithaca, N.Y., Aug.). Springer-Verlag, Berlin, Germany, 1995, pp. 234–250.

BURKE, M., SRINIVASAN, H., AND SWEENEY, P. F. 1996. A framework for evaluating space and time overhead for C++ object models. Tech. Rep. RC 20421, IBM Thomas J. Watson Research Center, (Mar.).

CALDER, B. AND GRUNWALD, D. 1994. Reducing indirect function call overhead in C++ programs. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, (Portland, Ore., Jan.). ACM Press, New York, N.Y., pp. 397–408.

CALDER, B., GRUNWALD, D., AND ZORN, B. 1994. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, *2*, 4 (Dec.), 313–351.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, *17*, 4 (Dec.), 471–522.

CARINI, P., HIND, M., AND SRINIVASAN, H. 1995. Type analysis algorithm for C++. Tech. Rep. RC 20267, IBM Thomas J. Watson Research Center.

CAUDILL, P. J. AND WIRFS-BROCK, A. 1986. A third generation Smalltalk-80 implementation. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, Ore., Sept.). *SIGPLAN Not.*, *21*, 11 (Nov.), 119–130.

CHAMBERS, C. 1992. *The Design and Implementation of the SELF Compiler.* Ph.D. thesis, Stanford Univ., (Mar.).

CHAMBERS, C. 1993. The Cecil language: Specification and rationale. Tech. Rep. TR-93-03-05, Dept. of Computer Science, Univ. of Washington, (Mar.).

CHAMBERS, C., DEAN, J., AND GROVE, D. 1996. Whole-program optimization of object-oriented languages. Tech. Rep. 96-06-02, Dept. of Computer Science, Univ. of Washington, (June).

CHAMBERS, C. AND UNGAR, D. 1989a. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, (Portland, Ore., June). *SIGPLAN Not.*, *24*, 7 (July), 146–160.

CHAMBERS, C. AND UNGAR, D. 1989b. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Portland, Ore., June). *SIGPLAN Not.*, *24*, 7 (July), 146–160.

CHAMBERS, C. AND UNGAR, D. 1991a. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. *LISP and Symbolic Computation*, *4*, 3 (July), 283–310.

CHAMBERS, C. AND UNGAR, D. 1991b. Making pure object-oriented languages practical. In *Proceedings of the 1991 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Phoenix, Ariz., Oct.). *SIGPLAN Not.*, *26*, 11 (Nov.), 1–15.

CHAMBERS, C., UNGAR, D., CHANG, B.-W., AND HOLZLE, U. 1991a. Parents are shared parts of objects: inheritance and encapsulation in SELF. *LISP and Symbolic Computation*, *4*, 3 (July), 207–222.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1991b. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *LISP and Symbolic Computation*, *4*, 3 (July), 243–281.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, (Charleston, South Carolina, Jan.). ACM Press, New York, N.Y., pp. 232–245.

CIERNIAK, M. AND LI, W. 1997. Briki: an optimizing Java compiler. In *Digest of Papers, Spring COMPCON 1997, Forty-Second IEEE Computer Society International Conference*, (San Jose, Calif., Feb.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 179–184.

COGGINS, J. M. 1991. Why does this program run so long? *C++ Report*, *3*, 6 (June), 21–24.

COGGINS, J. M. 1993. Speed of C++ vs. C: Myths, data, and skepticism. *C++ Report*, (Jan.), 25–27.

COOPER, K. D., HALL, M. W., AND KENNEDY, K. 1993. A methodology for procedure cloning. *Computer Languages*, *19*, 2 (Apr.), 105–117.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, *13*, 4 (Oct.), 451–490.

DAHL, O.-J. AND MYRHAUG, B. 1973. *Simula Implementation Guide.* Oslo, Sweden, (Mar.).

DAHL, O.-J. AND NYGAARD, K. 1966. Simula – an Algol-based simulation language. *Commun. ACM*, *9*, 9 (Sept.), 671–678.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1994. Identifying profitable specialization in object-oriented languages. In *Proceedings of the ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, (Orlando, Fla., June), pp. 85–96.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1995. Selective specialization for object-oriented languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (La Jolla, Calif., June). *SIGPLAN Not.*, *30*, 6, 93–102.

DEAN, J., DEFOUW, G., GROVE, D., LITVINOV, V., AND CHAMBERS, C. 1996. Vortex: an optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (San Jose, Calif., Oct.). *SIGPLAN Not.*, *31*, 10, 83–100.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In OLTHOFF, W., ED., *Proceedings of the Ninth European Conference on Object-Oriented Programming – ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, (Aarhus, Denmark, Aug.). Springer-Verlag, Berlin, Germany, pp. 77–101.

DEUTSCH, L. P. AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, (Salt Lake City, Utah, Jan.). ACM Press, New York, N.Y., pp. 297–302.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (San Jose, Calif., Oct.). *SIGPLAN Not.*, *31*, 10, 292–305.

DIXIT, K. M. 1991. The SPEC benchmarks. *Parallel Comput.*, *17*, 1195–1209.

DIXIT, K. M. 1992. New CPU benchmarks from SPEC. In *Digest of Papers, Spring COMPCON 1992, Thirty-Seventh IEEE Computer Society International Conference*, (San Francisco, Calif., Feb.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 305–310.

DIXON, R., MCKEE, T., SCHWEIZER, P., AND VAUGHAN, M. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of the 1989 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (New Orleans, Louisiana, Oct.). *SIGPLAN Not.*, *24*, 10, 211–214.

DRIESEN, K. 1993. Selector table indexing & sparse arrays. In *Proceedings of the 1993 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Washington, D.C., Oct.). *SIGPLAN Not.*, *28*, 10, 259–270.

DRIESEN, K. AND HÖLZLE, U. 1995. Minimizing row displacement dispatch tables. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Austin, Tex., Oct.). *SIGPLAN Not.*, *30*, 10, 141–155.

DRIESEN, K. AND HÖLZLE, U. 1996. The direct cost of virtual function calls in C++. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (San Jose, Calif., Oct.). *SIGPLAN Not.*, *31*, 10, 306–323.

DRIESEN, K., HÖLZLE, U., AND VITEK, J. 1995. Message dispatch on pipelined processors. In OLTHOFF, W., ED., *Proceedings of the Ninth European Conference on Object-Oriented Programming – ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, (Aarhus, Denmark, Aug.). Springer-Verlag, Berlin, Germany, pp. 253–282.

EDELSOHN, D. J. 1994. A generalized expression optimization hook for C++ on a high-performance architectures. In *Proceedings of IEEE Scalable High Performance Computing Conference*, (Knoxville, Tennessee, May). IEEE Computer Society Press, Los Alamitos, Calif., pp. 381–387.

ELLIS, M. AND STROUSTROUP, B. 1990. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Mass.

FEINBERG, N., KEENE, S. E., MATHEWS, R. O., AND WITHINGTON, P. T. 1997. *Dylan Programming: An Object-Oriented and Dynamic Language.* Addison-Wesley, Reading, Mass.

FERNANDEZ, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (La Jolla, Calif., June). *SIGPLAN Not.*, *30*, 6, 103–115.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification.* Addison-Wesley, Reading, Mass.

GRAVER, J. O. 1989. *Type-checking and Type-inference for Object-oriented Programming Languages.* Ph.D. thesis, Univ. of Illinois at Urbana-Champaign.

GRAVER, J. O. AND JOHNSON, R. E. 1990. A type system for Smalltalk. In *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages*, (San Francisco, Calif., Jan.). ACM Press, New York, N.Y., pp. 136–150.

GROVE, D., DEAN, J., GARRETT, C., AND CHAMBERS, C. 1995. Profile-guided receiver class prediction. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Austin, Tex., Oct.). *SIGPLAN Not.*, *30*, 10, 108–123.

GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. 1997. Call graph construction in object-oriented languages. In *Proceedings of the 1997 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Atlanta, Ga., Oct.). *SIGPLAN Not.*, *32*, 10.

HAMILTON, J. 1997. Montana smart pointers: They're smart, and they're pointers. In *Proceedings of the 1997 USENIX Conference on Object-Oriented Technologies and Systems*, (Portland, Ore., June). Usenix Association, Berkeley, Calif.

HARBISON, S. P. 1992. *Modula-3.* Prentice-Hall.

HENDERSON, R. AND ZORN, B. 1994. A comparison of object-oriented programming in four modern languages. *Softw. Pract. Exper.*, *24*, 11 (Nov.), 1077–1095.

HENNESSY, J. L. AND PATTERSON, D. A. 1990. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, Calif.

HÖLZLE, U. 1994. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* Ph.D. thesis, Stanford Univ., (Aug.).

HÖLZLE, U. AND AGESEN, O. 1995. Dynamic versus static optimization techniques for object-oriented languages. *Theory and Practice of Object Systems*, *1*, 3, 167–188.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In AMERICA, P., ED., *Proceedings of the European Conference on Object-Oriented Programming – ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, (Geneva, Switzerland, July). Springer-Verlag, Berlin, Germany, pp. 21–38.

HÖLZLE, U. AND UNGAR, D. 1994. A third-generation SELF implementation: reconciling responsiveness with performance. In *Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, OR, Oct.). *SIGPLAN Not.*, *29*, 10, 229–243.

HSIEH, C.-H. A., CONTE, M. T., JOHNSON, T. L., GYLLENHAAL, J. C., AND HWU, W.-M. W. 1997. A study of the cache and branch performance issues with running Java on current hardware platforms. In *Digest of Papers, Spring COMPCON 1997, Forty-Second IEEE Computer Society International Conference*, (San Jose, Calif., Feb.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 211–216.

HSIEH, C.-H. A., GYLLENHAAL, J. C., AND HWU, W. W. 1996. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, (Paris, France, Dec.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 90–97.

INGALLS, D. H. 1986. A simple technique for handling multiple polymorphism. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, Ore., Sept.). *SIGPLAN Not.*, *21*, 11 (Nov.), 347–349.

JOHNSON, R. E. 1986. Type-checking Smalltalk. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, Ore., Sept.). *SIGPLAN Not.*, *21*, 11 (Nov.), 315–321.

JOHNSON, R. E., GRAVER, J. O., AND ZURAWSKI, L. W. 1988. TS: an optimizing compiler for Smalltalk. In *Proceedings of the 1988 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (San Diego, Calif., Sept.). *SIGPLAN Not.*, *23*, 11 (Nov.), 18–26.

JONES, N. D., GOMARDE, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, N.J.

JONES, N. D. AND MUCHNICK, S. 1976. Binding time optimization in programming languages. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, (Atlanta, Ga., Jan.). ACM Press, New York, N.Y., pp. 77–94.

KAPLAN, M. A. AND ULLMAN, J. D. 1980. A scheme for the automatic inference of variable types. *J. ACM*, *27*, 1 (Jan.), 128–145.

KICZALES, G. AND RODRIGUEZ, L. 1990. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, (Nice, France, June). ACM Press, New York, N.Y., pp. 99–105.

KOZEN, D., PALSBERG, J., AND SCHWARTZBACH, M. I. 1994. Efficient inference of partial types. *J. Comput. Syst. Sci.*, *49*, 2 (Oct.), 306–324.

KROGDAHL, S. 1985. Multiple inheritance in Simula-like languages. *BIT*, *25*, 2, 318–326.

LAMBRIDGE, H. D. 1997. Java bytecode optimizations. In *Digest of Papers, Spring COMPCON 1997, Forty-Second IEEE Computer Society International Conference*, (San Jose, Calif., Feb.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 206–210.

LANDI, W., RYDER, B. G., AND ZHANG, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Albuquerque, New Mexico, June). *SIGPLAN Not.*, *28*, 6, 56–67.

LARCHEVEQUE, J. M. 1994. Interprocedural type propagation for object-oriented languages. *Sci. Comput. Programming*, *22*, 3 (June), 257–282.

LEA, D. 1990. Customization in C++. In *Proceedings of the 1990 USENIX C++ Conference*, (San Francisco, Calif., Apr.). Usenix Association, Berkeley, Calif., pp. 301–314.

LEE, Y. AND SERRANO, M. J. 1995. Dynamic measurements of C++ program characteristics. Tech. Rep. STL TR 03.600, IBM Santa Teresa Laboratory, (Jan.).

LINDHOLM, T. AND YELLIN, F. 1997. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Mass.

MEYER, B. 1992. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, N.J.

MEYERS, S. 1995. Writing efficient C++ programs. In *Object Expo Europe 1995 Objects Expo Europe '95*, (London, England, Sept.). *Object Expo Europe 1995*, pp. 186–187.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, *17*, 3 (Dec.), 348–375.

MORRIS, J. H. 1968. *Lambda-Calculus Models of Programming Languages*. Ph.D. thesis, Massachussets Institute of Technology.

MOSSENBOCK, H. AND WIRTH, N. 1991. The programming language Oberon-2. Tech. rep., Institue for Computer Systems, ETH, Zurich.

MYERS, A. C. 1995. Bidirectional object layout for separate compilation. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Austin, Tex., Oct.). *SIGPLAN Not.*, *30*, 10, 124–139.

NACKMAN, L. R. AND BARTON, J. J. 1994. Base-Class Composition with Multiple Derivation and Virtual Bases. In *Proceedings of the 1994 USENIX C++ Conference*, (Cambridge, Mass., Apr.). Usenix Association, Berkeley, Calif., pp. 57–71.

NAIR, R. 1994. Performance evaluation of C++ applications on the IBM RS/6000. Unpublished IBM Confidential Report, Nov. 1994.

NYGAARD, K. AND DAHL, O.-J. 1978. The development of the SIMULA languages. In *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*, (Los Angeles, Calif., June). *SIGPLAN Not.*, *13*, 8 (Aug.), 245–272.

OXHØJ, N., PALSBERG, J., AND SCHWARTZBACH, M. I. 1992. Making type inference practical. In MADSEN, O. L., ED., *Proceedings of the European Conference on Object-Oriented Programming – ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, (Utrecht, Netherlands, June). Springer-Verlag, Berlin, Germany, pp. 329–349.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-oriented type inference. In *Proceedings of the 1991 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Phoenix, Ariz., Oct.). *SIGPLAN Not.*, *26*, 11 (Nov.), 146–161.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1994a. *Object-Oriented Type Systems.* John Wiley and Sons, New York, N.Y.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1994b. Static typing for object-oriented programming. *Sci. Comput. Programming*, *23*, 1 (Oct.), 19–53.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1995. Safety analysis versus type inference. *Information and Computation*, *118*, 1 (Apr.), 128–141.

PANDE, H. D. AND RYDER, B. G. 1994. Static type determination for C++. In *Proceedings of the 1994 USENIX C++ Conference*, (Cambridge, Mass., Apr.). Usenix Association, Berkeley, Calif., pp. 85–97.

PANDE, H. D. AND RYDER, B. G. 1995. Static type determination and aliasing for C++. Tech. Rep. LCSR-TR-250, Dept. of Computer Science, Rutgers University, (July).

PANDE, H. D. AND RYDER, B. G. 1996. Data-flow-based virtual function resolution. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, pp. 238–254.

PLEVYAK, J. AND CHIEN, A. A. 1994. Precise concrete type inference for object-oriented languages. In *Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Portland, OR, Oct.). *SIGPLAN Not.*, *29*, 10, 324–340.

PLEVYAK, J. AND CHIEN, A. A. 1995. Type directed cloning for object-oriented programs. In HUANG, C.-H., SADAYAPPAN, P., BANERJEE, U., GELERNTER, D., NICOLAU, A., AND PADUA, D., EDS., *Proceedings of the Eighth International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, (Columbus, Ohio, Aug.). Springer-Verlag, Berlin, Germany, 1996, pp. 566–580.

PORAT, S., BERNSTEIN, D., FEDOROV, Y., RODRIGUE, J., AND YAHAV, E. 1996. Compiler optimizations of C++ virtual function calls. In *Proceedings of the Second Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada, June). Usenix Association, pp. 3–14.

PROEBSTING, T. A., TOWNSEND, G., BRIDGES, P., HARTMAN, J. H., NEWSHAM, T., AND WATTERSON, S. A. 1997. Toba: Java for applications – a way ahead of time (WAT) compiler. In *Proceedings of the 1997 USENIX Conference on Object-Oriented Technologies and Systems*, (Portland, Ore., June). Usenix Association, Berkeley, Calif.

PUGH, W. AND WEDDELL, G. 1990. Two-directional record layout for multiple inheritance. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (White Plains, N.Y., June). *SIGPLAN Not.*, *25*, 6, 85–91.

REYNOLDS, J. C. 1969. Automatic computation of data set definitions. In *Information Processing '68*. North-Holland Publishing Company, Amsterdam, pp. 456–461.

ROSE, J. R. 1988. Fast dispatch mechanisms for stock hardware. In *Proceedings of the 1988 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (San Diego, Calif., Sept.). *SIGPLAN Not.*, *23*, 11 (Nov.), 27–35.

RUF, E. AND WEISE, D. 1991. Using types to avoid redundant specialization. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, (New Haven, Conn., June). *SIGPLAN Not.*, *26*, 9 (Sept.), 321–333.

RYDER, B. G. 1979. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, *SE-5*, 3 (May), 216–226.

SCHWARTZBACH, M. I. 1991. Type inference with inequalities. In ABRAMSKY, S. AND MAIBAUM, T. S., EDS., *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 493 of *Lecture Notes in Computer Science*, (Brighton, England, Apr.). Springer-Verlag, Berlin, Germany, pp. 441–455.

SERRANO, M., 1997. Personal communication. IBM Santa Teresa Laboratory, 1997.

SESHADRI, V. 1997. IBM high performance compiler for Java: An optimizing native code compiler for Java applications. *AIXpert*, (Sept.).

SOROKER, D., KARASICK, M., BARTON, J., AND STREETER, D. 1997. Extension mechanisms in Montana. In *Proceedings of the Eighth Israel Conference on Computer Systems and Software Engineering*, (Herzliya, Israel, June).

SRINIVASAN, H. AND SWEENEY, P. F. 1996. Evaluating virtual dispatch mechanisms for C++. Tech. Rep. RC 20330, IBM Thomas J. Watson Research Center, (Jan.).

SRIVASTAVA, A. 1992. Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems*, *1*, 4 (Dec.), 355–364.

STEENSGAARD, B. 1996a. Points-to analysis by type inference of programs with structures and unions. In GYIMOTHY, T., ED., *Proceedings of the Sixth International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, (Linkoping, Sweden, Apr.). Springer-Verlag, Berlin, Germany, pp. 136–150.

STEENSGAARD, B. 1996b. Points-to analysis in almost linear time. In *Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages*, (St. Petersburg Beach, Fla., Jan.). ACM Press, New York, N.Y., pp. 32–41.

STROUSTRUP, B. 1986. *The C++ Programming Language.* Addison-Wesley, Reading, Mass.

STROUSTRUP, B. 1989. Parameterized types for C++. *Computing Syst.*, *2*, 1 (Winter), 55–85.

SUBRAMANIAN, S., TSAI, W.-T., AND KIRANI, S. H. 1994. Hierarchical data flow analysis for O-O programs. *Journal of Object-Oriented Programming*, *7*, 2 (May), 36–46.

SÜDHOLT, M. AND STEIGNER, C. 1992. On interprocedural data flow analysis for object oriented languages. In KASTENS, U. AND PFAHLER, P., EDS., *Proceedings of the Fourth International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, (Paderborn, Germany, Oct.). Springer-Verlag, Berlin, Germany, pp. 156–162.

SUZUKI, N. 1981. Inferring types in Smalltalk. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, Va., Jan.). ACM Press, New York, N.Y., pp. 187–199.

SWEENEY, P. F., 1997. Personal communication. IBM Thomas J. Watson Research Center, 1997.

SWEENEY, P. F. AND TIP, F. 1997. A study of dead data members in C++ applications. Tech. Rep. RC 21051, IBM Thomas J. Watson Research Center, (Nov.).

TENNENBAUM, A. 1974. Type determination for very high level languages. Tech. Rep. NSO-3, Courant Institute of Mathematical Sciences, New York Univ., New York, N.Y.

TIP, F., CHOI, J.-D., FIELD, J., AND RAMALINGAM, G. 1996. Slicing class hierarchies in C++. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (San Jose, Calif., Oct.). *SIGPLAN Not.*, *31*, 10, 179–197.

TYMA, P. 1996. Tuning Java performance. *Dr. Dobb's Journal*, *21*, 4 (Apr.), 52, 55–6, 58.

UNGAR, D. 1986. *Design and Implementation of a High-performance Smalltalk System.* MIT Press, Cambridge, Mass.

UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HÖLZLE, U. 1991. Organizing programs without classes. *LISP and Symbolic Computation*, *4*, 3 (July), 223–242.

UNGAR, D. AND SMITH, R. B. 1987. SELF: the power of simplicity. In *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, (Orlando, Fla., Oct.). *SIGPLAN Not.*, *22*, 12, 227–241.

UNGAR, D., SMITH, R. B., CHAMBERS, C., AND HOLZLE, U. 1992. Object, message, and performance: how they coexist in Self. *Computer*, *25*, 10 (Oct.), 53–64.

VITEK, J. AND HORSPOOL, R. N. 1994. Taming message passing: efficient method look-up for dynamically typed languages. In NIERSTRASZ, O. M., ED., *Proceedings of the Eighth European Conference on Object-Oriented Programming – ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, (Bologna, Italy, July). Springer-Verlag, Berlin, Germany, pp. 432–449.

VITEK, J. AND HORSPOOL, R. N. 1996. Compact dispatch tables for dynamically typed object oriented languages. In GYIMOTHY, T., ED., *Compiler Construction. 6th International Conference, CC'96. Proceedings Proceedings of CC: International Conference on Compiler Construction*, (Linkoping, Sweden, Apr.). Springer-Verlag, pp. 309–325.

VITEK, J., HORSPOOL, R. N., AND UHL, J. S. 1992. Compile-time analysis of object-oriented programs. In KASTENS, U. AND PFAHLER, P., EDS., *Proceedings of the Fourth International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, (Paderborn, Germany, Oct.). Springer-Verlag, Berlin, Germany, pp. 236–250.

WAND, M. 1987. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, *10*, 2 (June), 115–121.

WEISE, D., CONYBEARE, R., RUF, E., AND SELIGMAN, S. 1991. Automatic online partial evaluation. In HUGHES, J., ED., *Functional Programming Languages and Computer Architecture: 5th ACM Conference*, volume 523 of *Lecture Notes in Computer Science*, (Cambridge, Mass., Aug.). Springer-Verlag, Berlin, Germany, pp. 165–191.

WILCOX, C. R., DAGEFORDE, M. L., AND JIRAK, G. A. 1978. *MAINSAIL Language Manual*. Stanford Univ., (Oct.).

WU, P.-C. AND WANG, F.-J. 1996. On efficiency and optimization of C++ programs. *Softw. Pract. Exper.*, *26*, 4 (Apr.), 453–465.