Whole-Program Optimization of Object-Oriented Languages

by

Jeffrey Adgate Dean

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

Doctor of Philosophy

University of Washington

1996

Approved by ______________________________
(Chairperson of Supervisory Committee)

____

____

____

____

Program Authorized
to Offer Degree ______________________________

Date ______________________________

University of Washington

Abstract

Whole-Program Optimization of Object-Oriented Languages

by Jeffrey Adgate Dean

Chairperson of the Supervisory Committee: Professor Craig Chambers
Department of Computer Science and Engineering

This dissertation examines the use of whole-program optimization as a way of improving the performance of object-oriented programming languages. Although object-oriented programming conveys a number of software engineering benefits, heavy application of its trademark feature, dynamic dispatching, imposes severe performance penalties when programs are compiled using traditional compilation techniques. Several new techniques that rely on whole-program optimization are described, and these techniques substantially improve the performance of object-oriented programs written in Cecil, Java, C++, and Modula-3.

Among the new techniques is *class hierarchy analysis*, which provides the compiler with knowledge of the class hierarchy of the entire program. This is an especially important optimization, because it allows programmers to write their programs using dynamic dispatching for all operations, which preserves maximal flexibility and extensibility, but permits the compiler to optimize away this flexibility when it is unused by a particular program. *Exhaustive class testing* allows the compiler to optimize message sends that have a limited degree of polymorphism by inserting tests to partition the potential classes of objects that can appear at a call site. A *selective specialization* algorithm combines static information with dynamic profile data to determine where it is profitable to compile multiple, specialized versions of a single source routine. *Inlining trials* provide a means of making inlining decisions that take into account the effects of post-inlining optimizations. Whole-program optimization in an interactive programming environment is made possible through the use of *selective recompilation*, a technique for invalidating only compiled code that is affected by a programming change. The techniques described in the thesis have been implemented in the Vortex compiler, a whole-program optimizing compiler developed as part of this dissertation.

Whole-program not only speeds up existing features of object-oriented languages, but it also allows new features to be added to languages with little or no cost, enabling more general models of dispatching that result in more natural and uniform programming languages. The benefits of the optimization techniques described in this thesis are shown to already be important, and their importance is likely to only increase as people adopt more and more object-oriented programming styles.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Many people have contributed to making graduate career rewarding and enjoyable. First, and foremost, I want to thank my advisor, Craig Chambers. He taught me about compilers and about how to do research, and helped me to focus on the important issues when looking at a new problem. For helping me to take off my blinders and focus on the big picture, I owe him a great deal. He also helped me wear a path between the Chateau and the Atrium, where we held countless discussions over coffee and pastries. Susan Eggers, David Notkin, and Brian Bershad formed the rest of my committee. Their insightful comments and helpful advice definitely improved the quality of this dissertation, to the benefit of you, the reader.

Dave Grove, Anthony LaMarca, Neal Lesh, Ted Romer, Mike Salisbury, and (for a brief time) Wayne Ohlrich, my office mates for the past few years in the portable trailer we call home, have been the best group of friends one can imagine. I'm certain that my graduation was delayed by several months due to our constant discussions of nothing in particular (and to our dart-playing period), but it was definitely worth it: they made the time much more enjoyable. Dave, in addition to being a close friend, has been a great colleague in the Cecil project. He was always there to bounce ideas off of, to help me complain about one thing or another, and to share the ups and downs of research. Ruth Anderson, Sung-Eun Choi, Wayne Wong, and many others have also livened up my life, and I'll always have many fond memories of our times together.

Finally, it was only with the support of my loving wife, Heidi, and my wonderful daughter, Victoria, that I was able to accomplish this. Their constant love and support make all of this worthwhile.

Dedication

To my father, Andy, my wife, Heidi, and my daughter, Victoria, and in memory of my mother, Lee, for their constant love and support.

Chapter 1

Introduction

This thesis explores the use of whole-program optimization as a means of improving the performance ofprograms written in object-oriented languages.

Object-oriented programming languages include several features designed to support the development of flexible, reusable, and extensible software. *Inheritance* allows one abstraction to be developed using incremental modification or extension of an existing abstraction. *Dynamic dispatching*, also known as message passing, introduces a level of indirection between clients and implementations of abstractions, allowing a single piece of client code to manipulate multiple different implementations of an abstraction, even within the same execution of the program. The programming styles encouraged by these features have led to the development of libraries with rich inheritance hierarchies in such areas as basic data structures, graphics, and numerical computations. Object-oriented programming has achieved a large degree of success, due to the natural match of many object-oriented notions to the way that people think about problems and application domains.

Unfortunately, programs written using object-oriented language features are difficult to implement efficiently using traditional compilation techniques. Dynamic dispatching incurs both the direct cost of determining the appropriate routine to execute at run-time, as well as the indirect opportunity cost of preventing both inlining and the post-inlining optimizations that inlining enables. To make matter worse, the programming styles fostered by object-oriented programming tend to exacerbate these costs. Classes and encapsulation encourage smaller routines and more frequent procedure calls. Factoring and a desire to support future extensibility and refinement encourage heavier use of dynamic dispatching to select the appropriate implementation of an operation based on the run-time class of an object. It is unfortunate that heavy use of these features, while increasing the software engineering benefits of object-oriented programming, incurs higher runtime performance costs. This presents the programmer with a dilemma: should she use object-oriented features everywhere, preserving maximal flexibility but sacrificing performance? Or should she use object-oriented features only where their flexibility is required at the moment, providing better performance at the cost of programmer effort and reduced flexibility? The problem with the second approach is that as software evolves, its requirements and design goals change, and flexibility may be required in pieces of the software that were previously seen to be relatively stable (and were thus built using higher performance, but less flexible language constructs).

The goal of this thesis is to explore compiler and implementation techniques that circumvent this tradeoff: allow programmers to write programs in a way that preserves maximal flexibility, and have the compiler take care of eliminating or reducing the costs of this flexibility in order to provide reasonable performance. In particular, we focus on whole-program optimization as a key technique, meaning that the compiler has knowledge of the source code of the entire program, permitting many beneficial “closed world” assumptions that enable many of the optimization techniques described in this thesis. This “closed world” assumption is the important aspect of whole-program optimization, distinguishing it from other kinds of interprocedural optimization that do not require the whole program. As Chapter 2 will explain, the more traditional separate compilation model forces the compiler to make many conservative assumptions that are especially limiting for object-oriented languages and features. Whole-program optimization avoids these problems, but several questions remain about its potential:

- *What sort of optimizations are enabled by knowledge of the whole program?*
- *Can whole-program optimization be practical?*
- *How does the choice of programming language and programming style affect whole-program optimization?*

This thesis provides answers to these and many related questions. The contributions of the thesis are as follows:

- We develop several new optimization techniques that substantially improve the performance of object-oriented languages. These techniques are shown to scale to large programs, and the techniques are effective on both singly-dispatched and multiply-dispatched languages.
- We show that whole-program optimization can be practical, through the construction and everyday use of Vortex, an whole-program optimizing compiler for object-oriented

languages. Two supporting techniques, selective recompilation and inlining trials, are presented that are make whole-program optimization more usable in environments that place a strong emphasis on quick turnaround after programming changes, such as interactive development environments.

- We evaluate these techniques across a range of programming languages, including Cecil, C++, Java, and Modula-3, providing the first cross-language comparison of modern optimization techniques for object-oriented languages.

The remainder of this thesis is organized as follows:

- Chapter 2 reviews background material, motivates the use of whole-program optimization, and describes the general structure of the Vortex compiler.
- Chapter 3 describes *class hierarchy analysis* and *exhaustive class testing*, two techniques that eliminate a substantial percentage of dynamic dispatches by analyzing the class hierarchy of the program. These techniques are evaluated on a suite of benchmark programs written in Cecil, C++, Java, and Modula-3. The effectiveness of the techniques both in isolation and in conjunction with other optimization techniques for object-oriented languages is examined, and the impact of programming style and programming language on the effectiveness of the optimizations is explored.
- Chapter 4 describes *selective specialization*, a technique that combines static knowledge of the program's inheritance hierarchy with dynamic profile data to decide when it is profitable to compile multiple versions of a single source method, in order to eliminate dynamic dispatches.
- Chapter 5 describes a framework for representing intermodule dependencies introduced by optimizations such as class hierarchy analysis, and describes how the Vortex compiler manages these dependencies to perform *selective recompilation* after programming changes.
- Chapter 6 describes *inlining trials*, a way of making inlining decisions that takes into account the effects of post-inlining optimizations. The optimizations described in this thesis provide the compiler with many additional opportunities to apply inlining, and making good inlining decisions that balance improved performance against increased compile time and compiled code space becomes increasingly important.
- Chapter 7 discusses potential future work and offers some concluding thoughts.

Finally, this introduction would not be complete without acknowledging that much of the work in this thesis is the result of collaboration with other members of the Cecil group. In particular, the technique of class hierarchy analysis described in Section 3.2 grew out of discussions of with Craig Chambers and Dave Grove, and its implementation in the Vortex compiler was very much a team effort, with each of us implementing key parts of the required infrastructure. The selective recompilation dependency mechanism described in Chapter 5was also a group implementation effort.

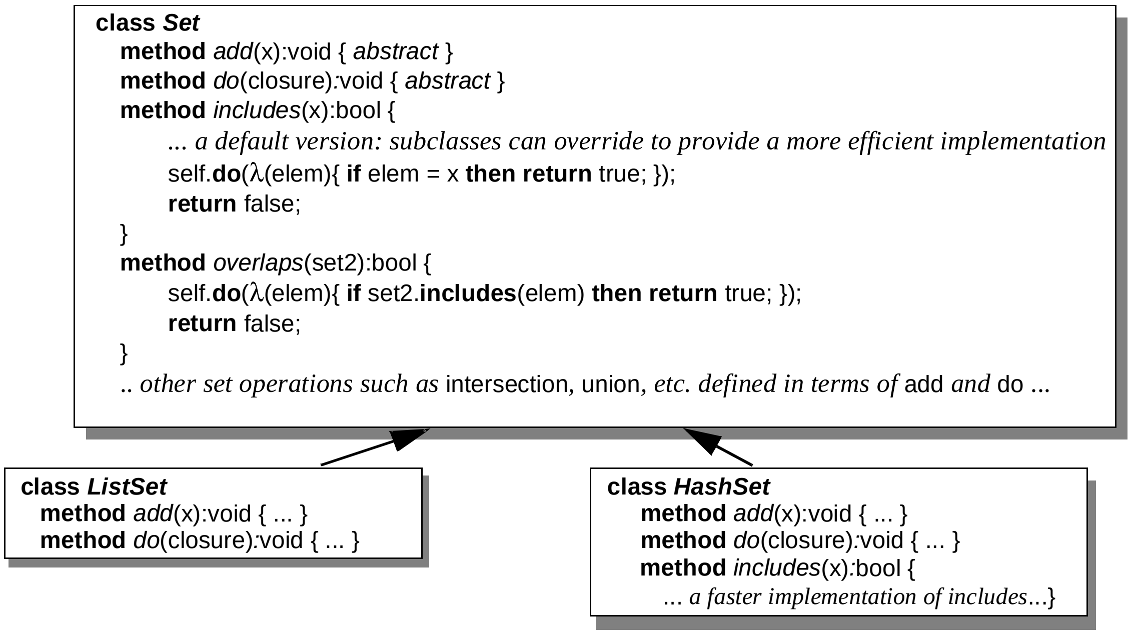
Chapter 2

Background

This chapter provides both an introduction to the concepts of object-oriented programming, and then goes on to discuss implementation techniques and issues for object-oriented languages. Section 2.1 discusses some common object-oriented programming idioms that lead to more flexible software. Section 2.2 describes why many of the language features used by these idioms lead to difficulties when producing efficient implementations of object-oriented languages. Section 2.3 provides some background on previously-developed optimization techniques for object-oriented languages. Section 2.4 discusses why separate compilation poses problems for the application of these optimizations, and Section 2.5 describes the Vortex compiler, the infrastructure developed for this thesis to explore whole-program optimization.

2.1 Object-Oriented Programming Style

A few simple ideas underlie object-oriented programming. One of the underpinnings is the use of *inheritance* to describe relationships among various classes in a hierarchy and to provide a means for reusing code. Common operations that apply to many classes can be *factored*, or defined higher up in the inheritance hierarchy, where their code can be reused by many subclasses. Each class can override its parent classes' operations, to provide its own refined behavior, and each operation invocation relies on *dynamic dispatching* to select the appropriate implementation of a behavior. To illustrate these concepts, consider the implementation of a set abstraction (a data structure near and dear to the hearts of compiler writers) shown in Figure 2.1. A basic set abstraction is defined in class Set. The Set class specifies that the add operation (which will add an element to a set), and the do operation (which will iterate over the elements in the set) are *abstract*, meaning that implementations of these operations will be provided by subclasses. Because the Set class specifies some methods as abstract, it is an abstract class: direct instances of the Set class cannot be created, since they would be lacking



```
class Set
  method add(x):void { abstract }
  method do(closure):void { abstract }
  method includes(x):bool {
        ... a default version: subclasses can override to provide a more efficient implementation
        self.do(λ(elem){ if elem = x then return true; });
        return false;
  }
  method overlaps(set2):bool {
        self.do(λ(elem){ if set2.includes(elem) then return true; });
        return false;
  }
  .. other set operations such as intersection, union, etc. defined in terms of add and do ...
```

```
class ListSet
  method add(x):void { ... }
  method do(closure):void { ... }
```

```
class HashSet
     method add(x):void { ... }
     method do(closure):void { ... }
     method includes(x):bool {
        ... a faster implementation of includes...}
```

Figure 2.1: A set abstraction hierarchy

the implementations of these abstract operations. Instead, the Set class provides a convenient place to define many other operations on sets, such as inclusion, union, intersection, and difference operations. These operations can be defined in class Set and rely on abstract operations such as add and do because such operations are invoked via dynamic dispatches: no matter what particular subclass of Set is in use, the implementation of these operations approriate to that representation will be invoked. Methods implemented in the set class and then inherited by its subclasses are referred to as *factored* methods, since they can be implemented just once, rather than repeatedly for each different kind of set implementation. Any subclasses of Set need only provide implementations of the abstract operations defined in class Set, and they can inherit the bulk of the operations on sets from these factored methods. Because the factored methods are implemented by sending messages to self to invoke other set operations (including the abstract methods defined in subclasses), the single version of each operation functions perfectly well when operating on any of the different kinds of subclasses, even permitting them to be mixed. For example, the implementation of overlaps can operate on a pair of ListSet objects, a pair of HashSet objects, or a mix of both. This style of programming, where much code is inherited from factored versions of methods, and where different implementations of classes can be mixed, simplifies many kinds of programming tasks. For example, defining a new kind of set abstraction (perhaps with different time complexities for the various set operations) is simply a matter of writing an operation to add an element to the set and an operation to iterate over the elements in the set. The rest of the set operations can be inherited from the abstract Set class, and furthermore, this new set operation can be mixed and propagated to others clients of the program that expect sets and can be manipulated without the client having to know the representation details of the set being manipulated.

2.2 The Costs of Dispatching

As the previous section described, object-oriented programming is one way of making it easier to reuse and extend existing software compoents, rather than always starting from scratch when developing a new component. Unfortunately, the characteristics of this programming style, and in particular the frequent use of dynamic dispatches, are a serious impediment to the efficient implementations of object-oriented languages, for two reasons. First, a dynamic dispatch incurs the *direct cost* of computing what method should be invoked, based on the run-time class of the arguments to the message send, and this cost is over and above the procedure call overhead of invoking the method. Second, and often more significant, is the *indirect opportunity costs* of preventing the application of traditional compiler optimizations, such as inlining. Without the kinds of analyses and transformations developed in this thesis and other recent research [Chambers 92, Hölzle 94], compilers must leave dynamic dispatches largely unoptimized, treating them as procedure calls to unknown routines (and adopting appropriate conservative assumptions about the behavior of these unknown routines). When dynamic dispatches are used infrequently, this does not significantly impact performance. Frequent use of dynamic dispatching, however, can have substantial performance implications. Modern architectures, such as the MIPS R10000 [Martin et al. 95] or the Intel Pentium Pro [Int95], exploit fine-grained parallelism by having a large window of ready instructions to issue and they rely on predictable control flow in order to keep this window full of useful instructions. In such systems, the frequent indirect control transfers associated with dynamic dispatches can be a barrier to performance [Calder & Grunwald 94, Driesen et al. 95] (although their impact can be somewhat mitigated through the use of hardware for predicting the targets of indirect jumps, such as branch target buffers [Lee & Smith 84, Perleberg & Smith 93]). Modern compilers also rely on having large blocks of instructions to work with, so that they can apply techniques such as trace scheduling [Fisher 81] and speculative execution [Mowry et al. 92], in an effort to maximize instruction-level parallelism. The presence of dynamic dispatches in programs can inhibit the effectiveness of these optimizations, since the compiler is often unable to hoist instructions above or sink instructions below the code for a dynamic dispatch (because the effects of the potentially-invoked methods are often unknown). In short, the direct and indirect costs of frequent dynamic dispatches are already significant and are likely to become more significant in future architectures and compilers.

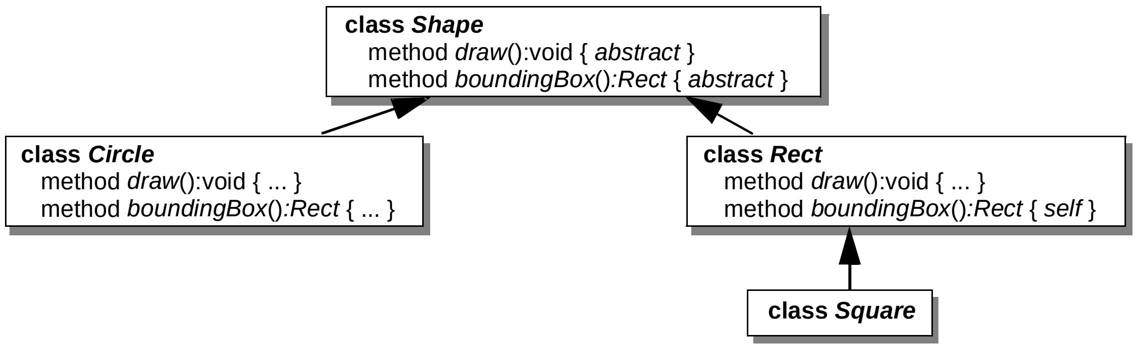
However, recent compilers for object-oriented languages have begun to optimize dynamic dispatches [Chambers et al. 89, Hölzle & Ungar 94, Chambers et al. 96]. If the compiler can deduce at compile-time that a particular dynamic dispatch will only invoke a single source method, then the code to perform the dynamic dispatch is unnecessary, and the dynamic dispatch can be converted to a statically-bound procedure call. Such statically-bound calls are then amenable to inlining. Failing this, the compiler can attempt to predict what classes are likely to appear at the message send and insert explicit tests for these cases. However, many of these techniques are not very effective under a separate compilation model, as will be discussed in Section 2.4.

2.3 Optimization Techniques for Object-Oriented Languages

The ultimate goal of most optimization techniques for object-oriented languages is to make dynamic dispatches execute quickly, preferably by eliminating them entirely. This is the goal of static class analyses, which are discussed in Section 2.3.1. If static class analysis is unable to eliminate the need for a dynamic dispatch, then inserting predictions about the likely behavior of the dynamic dispatch can improve performance by creating optimized code sequences for these cases. This is the realm of class prediction, and it is discussed in Section 2.3.2. If these predictions fail or cannot be applied, then the final alternative is to make the remaining dynamic dispatches execute as quickly as possible. Fast dispatching techniques are discussed in Section 2.3.5.

2.3.1 Static Class Analysis

There are many variants of static class analysis, including intraprocedural [Chambers & Ungar 90] and interprocedural analyses of varying complexity [Palsberg & Schwartzbach 91, Oxhøj et al. 92, Agesen et al. 93, Plevyak & Chien 94, Pande & Ryder 94, Agesen 95, Dean et al. 95b, Grove 95, Fernandez 95, Bacon & Sweeney 96, Diwan et al. 96], but they all share a


class *Shape*
method *draw*():void { *abstract* }
method *boundingBox*():*Rect* { *abstract* }
class *Circle*
method *draw*():void { ... }
method *boundingBox*():*Rect* { ... }
class *Rect*
method *draw*():void { ... }
method *boundingBox*():*Rect* { *self* }
class *Square*

```
let s:Shape := new Square(...);
let bb:Rect := s.boundingBox;
bb.draw;
```

Figure 2.2: Example class hierarchy and code fragment

common goal of narrowing the potential target methods of a message send by determining the potential classes of variables that are used as arguments to the send. At a message send, using the computed information about the possible classes, the compiler can perform method lookup at compile time to determine the set of potentially-invoked methods. In the best case, the result of this lookup will indicate that only a single method can be invoked, enabling the code for the message send to be replaced with a direct procedure call to the target method. Such a message send is said to be *statically-bound*, and is then amenable to further optimization by inlining. The remainder of this section illustrates how intraprocedural class analysis works, by means of the example class hierarchy and code fragment shown in Figure 2.2. In this example, a hierarchy of shapes have been defined, with each different kind of shape implementing the draw and boundingBox messages. The code fragment creates a Square object, and computes and draws its bounding box. The variables s and bb both admit subclass polymorphism, meaning that, for example, a variable of any class that inherits from class Shape could be assigned to variable s. If no static analysis is performed to determine the target of these messages, then there will be two dynamic dispatches executed in this code sequence: a send of boundingBox and a send of draw.

By analyzing the flow of classes through this code fragment we can determine the target methods of these two message sends. Figure 2.3 illustrates this analysis for the example code fragment, showing three different snapshots of the control flow graph for this example during the class analysis optimization phase. In Figure 2.3 (a), static class analysis has determined that after the new statement s must hold an instance of class Square. The analysis then proceeds to the send of the boundingBox message. At this point, the compiler performs compile-

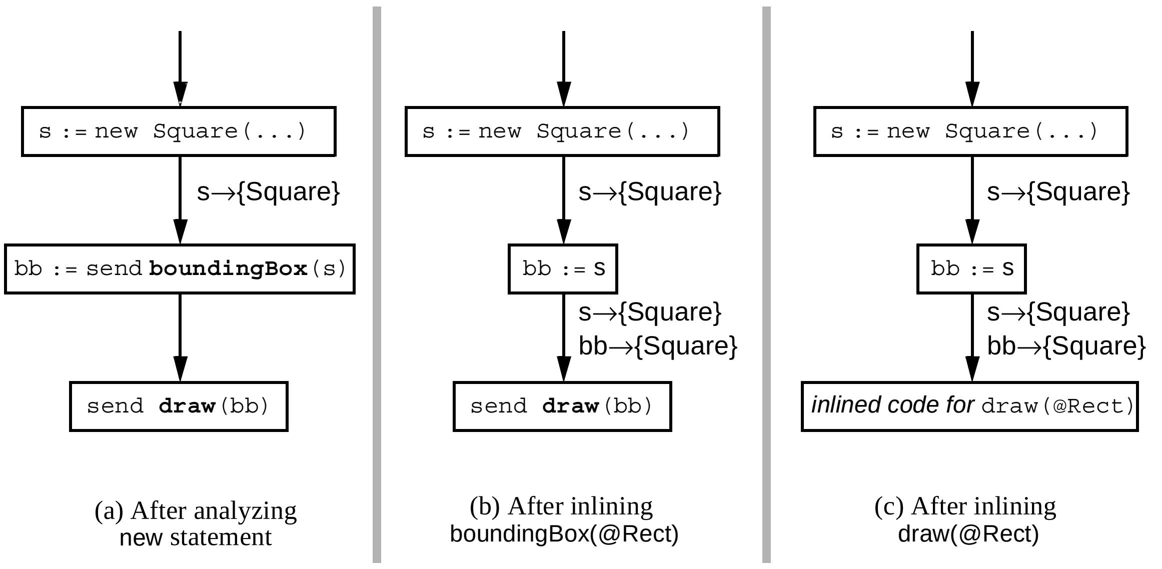


Figure 2.3: Effects of intraprocedural class analysis

time method lookup to determine what methods could be invoked when sending boundingBox to an instance of class Square, and finds that only boundingBox(@Rect) could be invoked by this send.[1] Since this method is a promising candidate for inlining, the compiler inlines its body in place of this statically-bound message send. If the method was not considered profitable to inline, the message send would have been replaced with a direct procedure call to this method, at least eliminating the direct overhead of the dispatch. In this case, however, the method is considered a profitable inlining candidate and class analysis continues with the now inlined code, as shown in Figure 2.3 (b). Performing inlining in this case also provides the additional indirect benefit of learning that bb also holds an instance of class Square. Analysis then continues with the send of draw(bb), and compile-time method lookup finds that the method draw(@Rect) will be invoked, enabling this message send to also be statically-bound and the body of the routine inlined, as shown in Figure 2.3 (c).

If the message send of boundingBox had not been inlined, an intraprocedural analysis would have learned nothing about the class of bb, and therefore would have been unable to statically-bind and inline the send of draw. For this reason, inlining is usually performed in parallel with static class analysis. This allows class analysis to be applied to the inlined code,

1. The next section discusses some programming languages such as Cecil, for which the compiler would be unable to resolve this method lookup at compile-time without whole-program knowledge, even though it knows the exact class of the receiver.

improving downstream class information both within the inlined code and farther downstream in the calling method, due to improved information about the return values of the method. In systems that implement only intraprocedural class analysis, this requirement that code be inlined in order to be analyzed increases the pressure to inline to obtain more class information [Chambers 92].

The compile time required for a purely intraprocedural class analysis scales well with program size since procedures tend to not increase in size as program size increases. The drawback is that there are not very many useful sources of class information within a single procedure, and so the performance benefits are often quite small. The primary sources of information include:

- *Object creation points*, as in the first line of the example in Figure 2.2 where the exact class[2] of the object being created is known.
- *Literal expressions* in some languages where the literals are first-class objects (e.g. ‘5’ is a SmallInt object in Self, Smalltalk and Cecil).

No useful information is known about the classes of the arguments to a method, including the receiver argument. All that is known about the receiver argument, for instance, is that it is some subclass of the class on which the method is specialized. This information is not useful for statically-binding messages, however, since other modules could define arbitrary subclasses that override any of the method definitions seen for this class. Without more specific information the compiler has no idea what method will be invoked when it has class information of this form. One way of obtaining more-specific information is to apply customization [Chambers & Ungar 89], in which a specialized version of a method is compiled for each class that inherits the method. This “seeds” intraprocedural class analysis with information about the exact class of the receiver argument, allowing all sends to the receiver to be statically-bound. Unfortunately, as discussed in Chapter 4, customization also dramatically increases compile time and compiled-code space requirements and does not scale well to large programs.

2. This thesis often uses the term *class*. In prototype-based languages, such as Cecil and Self, which do not have classes, this term is not strictly accurate. For these languages, the analogous concept is all objects in a single clone family. However, from the compiler’s point of view, both of these concepts are the same.

Extending class analysis to the interprocedural level is another way of obtaining more precise information about the possible classes stored in program variables. The most precise results can be obtained by flow-sensitive and context-sensitive interprocedural algorithms that track the flow of classes and values through the entire program, constructing a call graph in parallel with the class analysis process. Such algorithms are an active area of research [Palsberg & Schwartzbach 91, Oxhøj et al. 92, Agesen et al. 93, Plevyak & Chien 94, Agesen 95, Grove 95, Pande & Ryder 94], but these approaches have not yet been shown to scale well to programs of more than a few thousand lines. Furthermore, these algorithms are not incremental: any change to the program requires the reanalysis of the entire program, making them impractical for environments in which fast turnaround after a programming change is a significant consideration. Chapter 3 of this thesis presents class hierarchy analysis, an intermediate point between purely intraprocedural class analysis and more expensive, flow-based interprocedural class analyses, that can obtain significantly better results than just intraprocedural analysis, that scales to large programs, and can be applied in an incremental compilation environment.

2.3.2 Class Prediction

If static class analysis is unable to uniquely determine the target method of a particular message, the compiler can still try to optimize the message for the expected receiver classes. Given some receiver class frequency distribution for the message send site, the compiler can elect to insert a series of class tests for the most common receiver class(es), where each test branches upon success to a statically-bound and possibly inlined version of the message. A dynamically-dispatched version of the message remains to handle the situation where none of the tests succeeds at run-time. For example, in the control flow graph fragment shown in Figure 2.4, tests for the two most-common receiver classes at some call site have been inserted.

Performance when the receiver is one of the predicted classes improves as long as the benefits of avoiding the dynamic dispatch or of optimizing the callee in the context of the caller are greater than the run-time cost of a class test and all earlier failed tests. If the tested classes are sufficiently common, then the benefits to speeding up those tested cases will outweigh the slowdowns due to failed class tests incurred by the mispredicted cases. This issue is discussed further in Section 3.3.4.

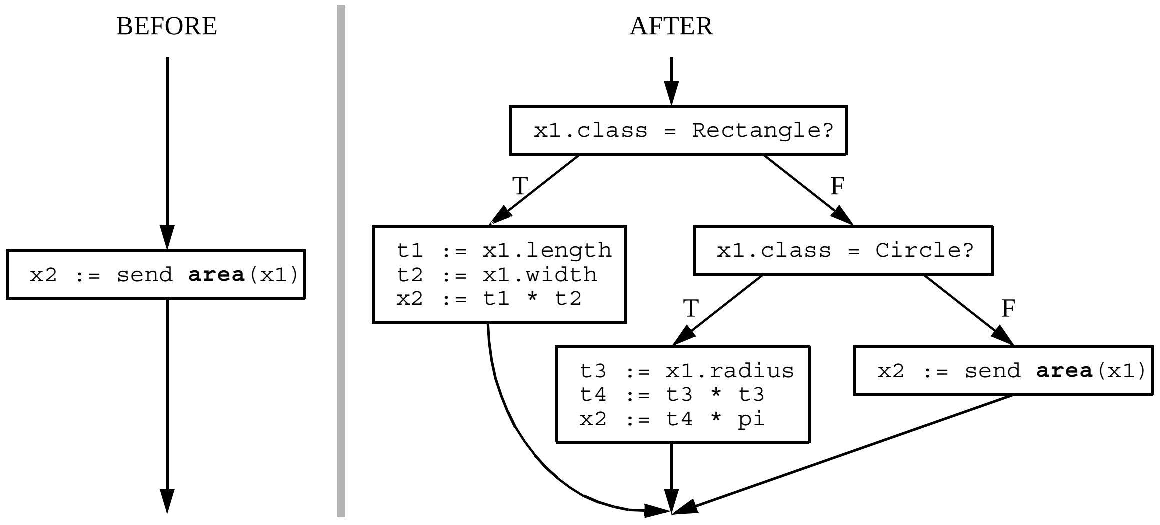


Figure 2.4: Control flow graph before and after insertion of class prediction tests

Static class information complements receiver class distribution information. Static class information provides an upper bound on the set of possible classes of a message's receiver, while receiver class distributions provide a kind of lower bound, indicating the set of classes that are expected. The two sources of information can be used in combination, filtering the set of predicted classes to include only those that are deemed possible by the static analysis.

Earlier compilers for dynamically-typed pure object-oriented languages. such as Smalltalk [Deutsch & Schiffman 84] and Self [Chambers et al. 89], incorporated a hard-wired table giving expected receiver classes for a number of commonly-occurring message names. For example, in Smalltalk, the + message was expected to be sent to `SmallInt` instances and `ifTrue:ifFalse:` to `True` or `False` instances. Compilers for other dynamically-typed languages such as Scheme often incorporate similar optimizations to speed the performance of generic arithmetic [Kranz et al. 86].

Receiver class distributions derived from dynamic profile information can greatly improve upon hard-wired receiver class distributions [Calder & Grunwald 94, Hölzle & Ungar 94, Grove et al. 95]. First, they more accurately reflect actual program behavior rather than hard-wired estimates built into the compiler before the program was written. Second, all messages in the program can have distribution information, instead of just a select few. This is important in encouraging programmers to define their own abstractions and messages: if programmers know that only the “built-in” operations are well-optimized, as is the case with

Smalltalk, then they will be more likely to write lower-level code using only primitive (but fast) operations instead of defining and using higher-level (but slower) abstractions where they are more natural. Third, distributions derived from profiles can exploit the increased levels of context provided by call-site-specific monitoring of receiver classes to be more precise than simple message summary distributions. For example, in most of the program a + operation will apply to `SmallInt` objects, but in a floating-point-intensive part of an application, call-site specific distributions can flag those + messages that are likely to have `SmallFloat` arguments.

Profile-guided class prediction can be applied in either an off-line or an on-line fashion. The off-line approach compiles the program statically using a profile gathered from earlier executions of the program. This approach works well if the profiles from earlier runs are good predictors of the behavior of future executions of the program. Our experience is that profile-guided receiver class distributions are reasonably stable across program runs [Grove et al. 95], but this may not be true of some programs. The latest Self compiler [Hölzle & Ungar 94] avoids this issue by gathering profile data in an on-line manner, and making use of the data during dynamic compilation. The system gathers receiver class distribution information as the program runs, and the system dynamically recompiles and reoptimizes frequently-executed procedures using the class distributions gathered so far for that particular program execution. This approach eliminates the need for a separate off-line training run and has the potential for producing better-optimized programs if different executions have substantially different profiles. It does, however, incur the costs associated with dynamic compilation, as discussed further in Section 2.3.4.

2.3.3 Splitting

Class prediction, whether hard-wired or profile-guided, can introduce many class test branches into the program. If a variable is sent several messages in sequence or is sent a message within a loop, these class tests can be seen as redundant. To avoid redundant and partially redundant tests, the control flow graph of the method being compiled can be modified by *splitting* the path between the merge following one occurrence of a class test and the next occurrence of the same test, as illustrated in Figure 2.5 [Chambers & Ungar 90].

One simple algorithm, implemented in the Vortex compiler, is to perform splitting in a lazy fashion. During forward static class analysis, the compiler tracks which variables have

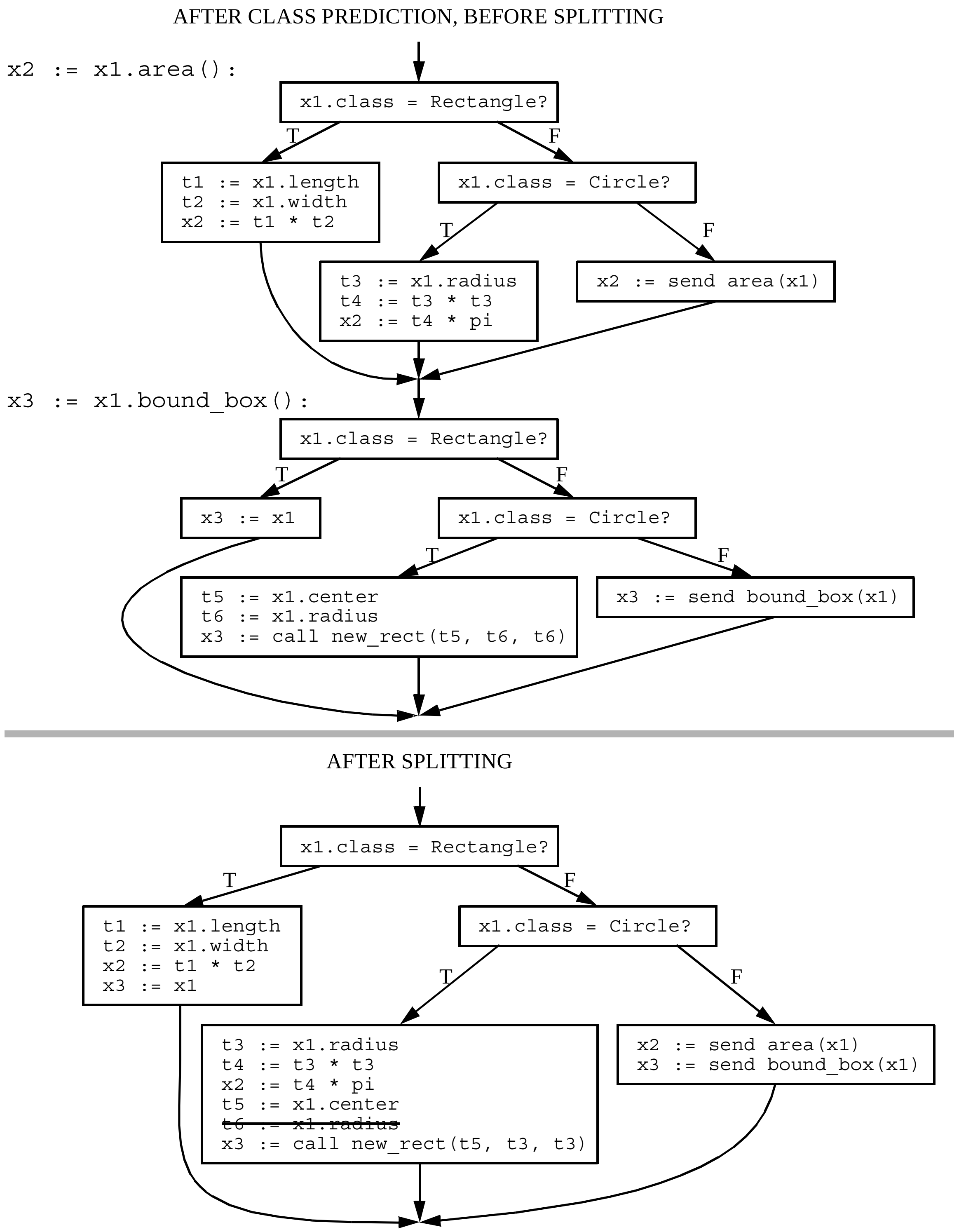


Figure 2.5: Control flow graph before and after splitting to eliminate redundant tests

had the information known about their potential classes "diluted" at merges, i.e., where the set of possible classes of a variable is larger after the merge than along one of its predecessors. At the point where a class test is encountered (or is about to be inserted as part of receiver class prediction), the compiler checks whether the tested variable has had its class information diluted by an earlier merge. If so, then the compiler scans back through the control flow graph to identify the paths back from the test to merge predecessors that would statically resolve the test, if any, and also accumulates a cost in terms of compiled code space for the potential splitting. Splitting is performed if there are paths that could be split off that would resolve the class test statically and the cost of the split is below a fixed threshold. An alternative approach to implementing splitting could apply a reverse dataflow analysis to identify "anticipatable class tests," propagating back from class tests to the merges that would resolve them. This approach would be similar to the one taken by Mueller & Whalley to avoid conditional branches [Mueller & Whalley 95].

To simplify its analysis, the current Vortex compiler does not support splitting past a loop entry or exit node. However, it is possible and often useful to split paths around loops, potentially splitting off whole copies of loops optimized for particular classes; early Self compilers implemented this optimization [Chambers & Ungar 90]. Loop splitting sometimes has the effect of peeling off the first iteration of the loop to hoist the invariant class tests out of the loop, as shown in Figure 2.6.

2.3.4 Dynamic Compilation

Deferring compilation until run-time is another useful technique for obtaining information about the behavior of a program, and has been used to good effect in a variety of ways in compilers for Smalltalk [Deutsch & Schiffman 84] and Self [Chambers & Ungar 91, Hölzle & Ungar 94]. Performing compilation at run-time has a number of advantages. First, code generation can be done lazily, by waiting until a piece of code is actually needed before generating it. This can be done either on a method-level granularity, by compiling a version of the method the first time it is invoked, or it can be finer-grained, as with the Self compiler's technique of lazy compilation for uncommon branches, in which compilation of certain unexpected paths through the control flow graph is delayed until the program actually reaches the path. Performing code generation lazily has the effect of providing faster turnaround time after programming changes, since the compiler only compiles parts of the program as they are needed. A second advantage of dynamic compilation is that dynamic information about the behavior of

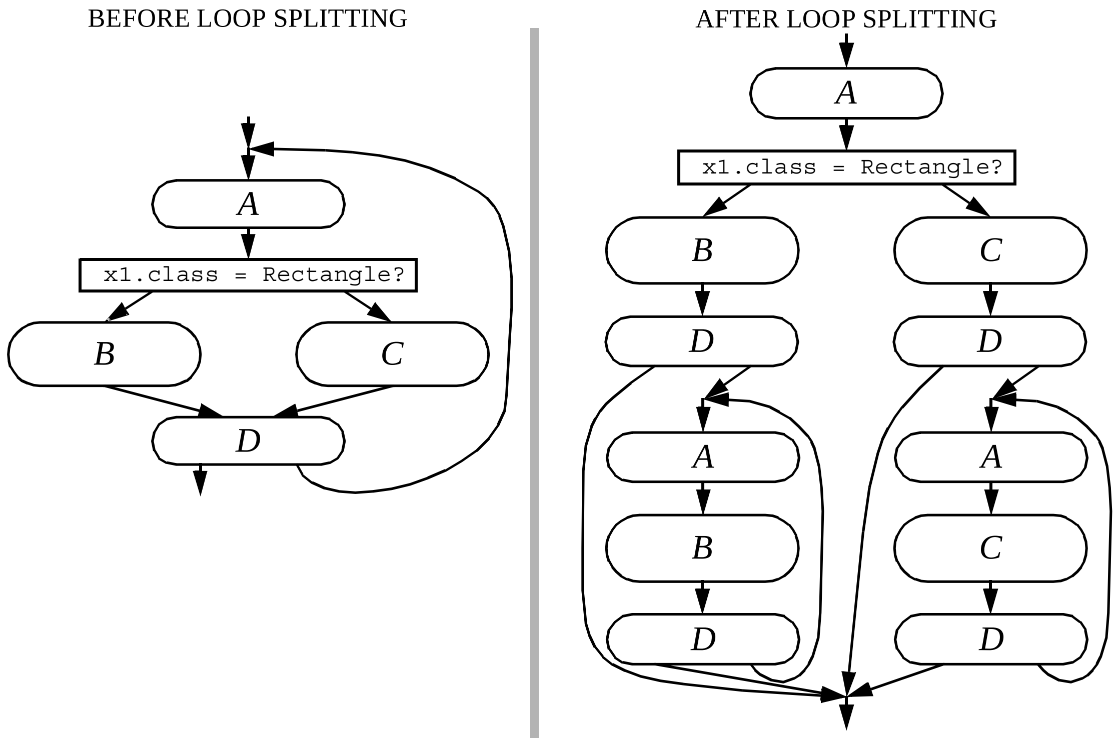


Figure 2.6: Loop splitting to create multiple loop bodies

the program can be put to use in the optimization process. The Self-91 compiler did this by initially compiling unoptimized code for a method, but with an execution frequency counter installed. When the counter reached a certain threshold, the code for the method would be recompiled with optimization. This allowed optimization effort to only be expended for the program's hot spots [Chambers 92]. The Self-93 compiler extended this idea by using profile-guided receiver class prediction [Hölzle & Ungar 94] to optimize each call site in the program with class predictions for just those classes that were frequently occurring at the call site.

Dynamic compilation is also useful for a number of other compilation and programming environment issues that are largely unrelated to this thesis. For example, dynamic compilation can help in the debugging of optimized code: instead of trying to provide the debugger with information of how the compiler's optimizations have affected the program's state, dynamic compilation can be used to shield the debugger from this complexity by dynamically *deoptimizing* code, allowing the debugger to assume that it is working with unoptimized code [Hölzle et al. 92]. Dynamic compilation is also useful for translating from machine independent intermediate forms such as Java byte codes [Gosling et al. 96] and Omnicode [AT et al. 96].

There are two main disadvantages of dynamic compilation. The first is that the language runtime system must include both a compiler and some representation of the program (source or intermediate code). The space concerns of these requirements may be somewhat mitigated by the fact that dynamic compilation can compile code lazily and can discard infrequently-used code, sometimes resulting in an overall space savings compared with a system that compiles all code statically. The more serious problem is that, because dynamic compilation is done during program execution, then any time spent during compilation is time that the program is not spending performing the actual computation. Because of this, the time actually spent performing compilation must be kept to a minimum, and as a result, most dynamic compilers perform only simple optimizations that do not require extensive analysis. Dynamic compilation will only improve performance if the benefits of dynamic information outweigh the benefits of the more powerful but time-consuming static analyses and optimizations that cannot be applied.

2.3.5 Efficient Dispatching

Eliminating dynamic dispatches through static analysis or optimizing them through the use of class predictions are the most effective way of improving the performance of dynamic dispatches, since they can get rid of most or all of the direct cost of dispatches, and they permit inlining, to eliminate much of the indirect costs of dispatches. Nonetheless, some dispatches are not able to be optimized through static analysis (either because the dispatch may actually invoke multiple target methods during the execution of the program or because the static analysis is not sufficiently powerful to prove that it will not do so) and are not amenable to the insertion of predictive tests (either because insufficient information about the likely behavior of the dispatch is available, or because the available information indicates that there are no cases that are sufficiently more likely than other cases to warrant the insertion of predictive tests). Making these dispatches, which are often heavily polymorphic, execute as fast as possible is important for obtaining high performance language implementations. A large body of research has devised techniques for dispatching messages efficiently, often in just a few instructions. These can be divided into two main categories: dispatch-table based techniques, and caching-based techniques. This section presents brief summaries of both these approaches. Driesen et al. present a more complete discussion of these techniques in the context of an exploration of their cost on modern superscalar processors [Driesen et al. 95].

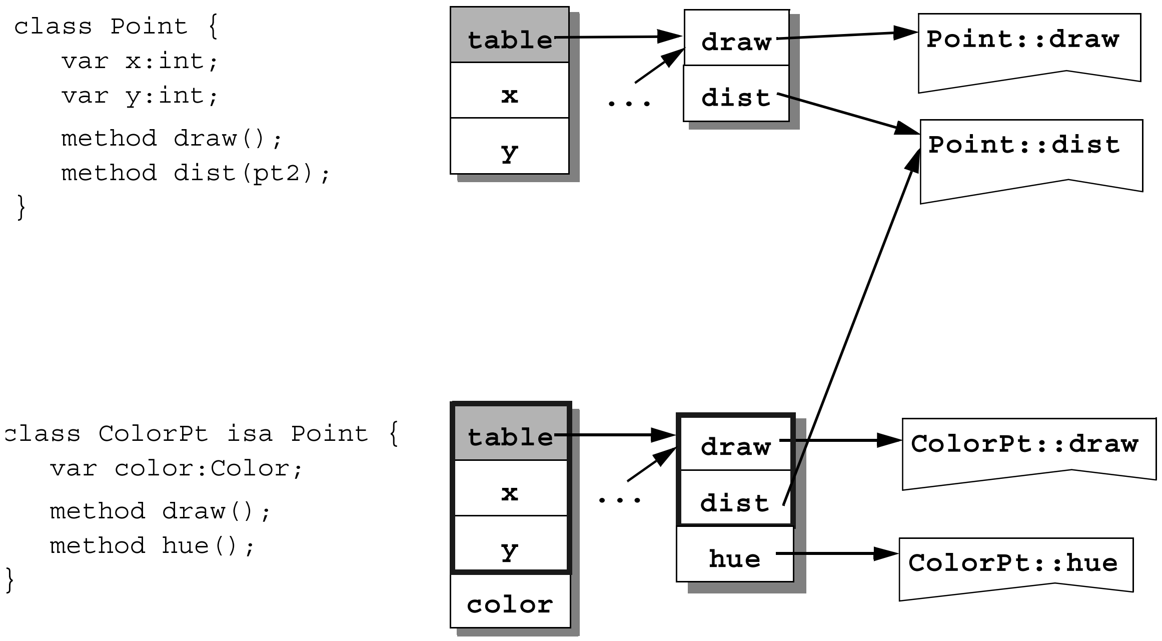


Figure 2.7: Example of dispatch tables

Dispatch tables were first used in Simula [Dahl & Myhrhaug 73] and today are used in most if not all implementations of C++ and Modula-3 [Stroustrup 91, Nelson 91]. The basic principle is to allocate a fixed index for each message that a class understands, in such a way that a message will have the same index in all subclasses. For each class, a table is allocated whose contents are pointers to the appropriate routine to invoke for that message for that class. Instances of the class all contain a pointer to the base of this table, so dispatching to the appropriate routine given an instance of a class involves (a) loading the address of the dispatch table from the object, (b) loading the address of the method to invoke from the appropriate index in the dispatch table, where the index is determined by the particular message being sent, and (c) calling the method at this address, usually by jumping indirectly through a register. This is illustrated in Figure 2.7. Dispatch tables are an attractive choice for implementing method dispatching for statically-typed languages because they provide constant-time dispatching of messages. The presence of multiple inheritance complicates the generation and use of dispatch tables somewhat, sometimes involving the use of different dispatch tables for each static view of a class [Stroustrup 91, Myers 95].

For dynamically-typed languages dispatch tables usually are not used. Because any message can in principle be sent to any object (with the possibility that a message not understood error may occur), the straightforward implementation of dispatch tables consumes *O(# of mes-*

sages $\times$ *# of classes)* space, and for large systems, this can be several megabytes [Driesen et al. 95]. Instead, many systems give up the constant-time dispatching property that dispatch tables provide, and use dynamic techniques that rely on some form of caching, with entries added to the cache lazily. The schemes differ in where the cache is located and in how large the cache can become. Early Smalltalk systems relied on a global lookup cache [Deutsch & Schiffman 84], which was a hash table, keyed by some hash function that combined the message name and the object's class. A dispatch consists of computing the hash key, and consulting the hash table. A successful search will return the address of the appropriate method to invoke. On an unsuccessful search, the method to invoke is computed by performing method lookup at run-time, and the result is then stored in the hash table.

It is often the case that the number of different receiver classes actually appearing at a particular call site is very small, even though the potential number of classes at a call site might be very large. This observation led to the development of *inline caching* [Deutsch & Schiffman 84], which is essentially a per-call-site, one-entry cache of the class of the last object appearing at the call site and the method to invoke for this class. A quick check is made to see if the class is the same as the last time through the call site. If so, then the system executes the method that was invoked last time. If the class is different, then the system falls back on a more expensive lookup technique, such as a global lookup cache, and changes the one-entry cache to reflect this new lookup. Typically, the cache is implemented by modifying the actual call instruction at the call site to invoke the new method, and each method has the test to check for a cache hit in the prologue code of the method.

Polymorphic inline caches (*PICs*) [Hölzle et al. 91] extend the idea of inline caches to use a small cache of previous lookup results (a typical maximum size is 10 elements). This cache is searched linearly, looking for the class to which the message is being sent. On a cache hit, the cache provides the address of the method to invoke. On a cache miss, a more expensive lookup technique is used (often a global lookup cache), and an entry added to the cache to reflect this new lookup. PICs can be implemented using a data representation of the cache, such as a linked list of cache entries, where lookup consists of iterating over this list, interpreting the data in the cache entry to implement the test. A faster implementation of PICs uses run-time code generation to generate a stub routine for each PIC that implements the cache lookup using inline tests against constant values, rather than interpreting the PIC data structure at run-time. In effect, the PIC stub routine represents the partial evaluation of the general-purpose PIC lookup routine with the current state of the cache. The generation of the stub routines is typically done by a hand-crafted code generator; recent developments in more automated techniques to apply run-time code generation could make this process easier [Auslander et al. 96, Knoblock & Ruf 96]. Some PIC implementations also utilize a move-to-front optimization on a cache hit, in an effort to exploit temporal locality in the stream of classes appearing at the call site. PICs perform better than inline caches when a call site has a limited degree of polymorphism. They also provide a convenient way of collecting class distributions for use in class prediction, since incrementing a counter in each cache entry is all that is required at each dispatch to gather the appropriate information [Hölzle 94]. In some respects, PICs implemented with generated code bear a striking resemblance to a sequence of inline class tests, except for the fact that the tests are out-of-line in a separate stub routine, and therefore inlining cannot be applied.

Dispatching for languages that support multi-methods, where the method to invoke can depend on the classes of any or all of the arguments to the message, rather than on just a single receiver argument position, provide additional challenges for implementing dispatches. Caching techniques and various forms of compressed dispatch tables have both been proposed as ways of implementing efficient message dispatching in languages that support multi-methods. Kiczales and Rodriguez propose a chained series of hash tables, one per dispatched argument position [Kiczales & Rodriguez 89], while Amiel et al. [Amiel et al. 94] and Chen et al. [Chen et al. 94] describe two different techniques to compress dispatch tables, but none of this work provides empirical data on the effectiveness of their technique for real programs. In the Vortex compiler, we use a straightforward extension of PICs [Hölzle et al. 91] to handle multi-methods, by testing all argument positions for the generic function that might affect the result of the method lookup.

2.4 Extensibility and Compilation Models

Most compilers, for languages as varied as C, C++, Fortran, Modula-3, and ML, rely on a separate compilation model, where a single module at a time is compiled. Although this model has several benefits, it does impose fairly severe restrictions on the optimization of many features of object-oriented programs. In large part, this is due to the ability in object-oriented languages to extend the behavior of an abstraction separately from the definition of an abstraction (for example, by defining a subclass and overriding the behavior of one of the inherited methods). Under separate compilation, illustrated in Figure 2.8, the programmer invokes the compiler separately on each of the individual source modules that make up the

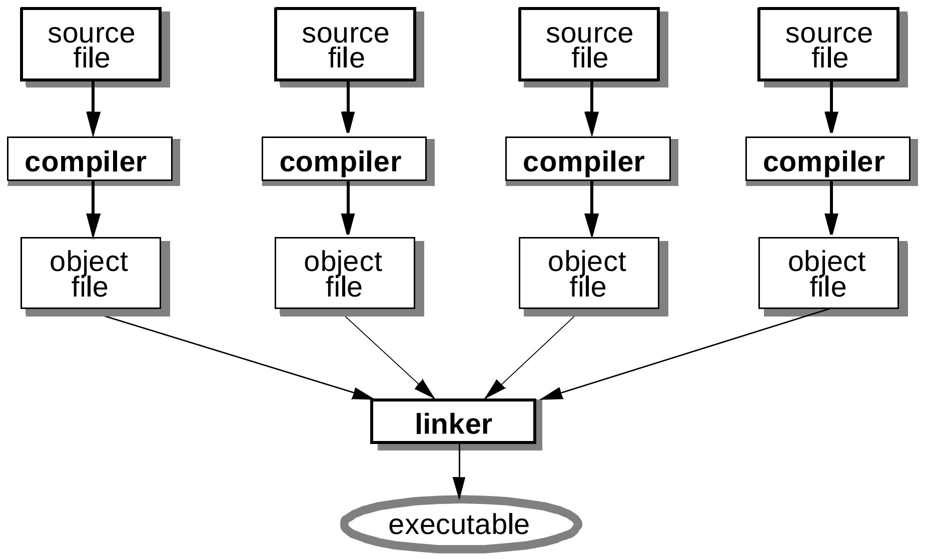


Figure 2.8: Separate compilation model

program. The compiler generates object code for each of these source modules in isolation, making conservative assumptions about the behavior of the other source modules in the program. Finally, the object code modules that make up the program are combined together by a linker to form the program executable. Unfortunately, under separate compilation, the compiler has to make conservative assumptions about the behavior and structure of the other modules in the program, and these assumptions hamper the ability to perform optimizations. Table 2.1 describes several kinds of extensibility that are allowed in different object-oriented languages, and discusses the implications for optimization under separate compilation of having to assume that this extensibility will occur. It also which of six object-oriented languages have language features that allow this kind of extensibility (and thus which languages have the accompanying implementation difficulties under separate compilation).

The key issue is that the mere potential for extensibility imposes severe restrictions on the kinds of optimizations that can be used, even though in actual practice, much of this potential extensibility is unused. An implementation that examines the whole program and compiles for the actual case, rather than the worst case, can generate significantly better code, as this thesis demonstrates.

One approach for delaying optimizations until the whole program is available is to perform optimizations at link-time, when a representation of the entire program is available (although dynamically-linked shared libraries can imply that even waiting until link-time is

Table 2.1: Extensibility and its implications

Kind of extensibility	**Implementation implications under separate compilation**	**Cecil**	**Dylan**	**CLOS**	**Java**	**Modula-3**	**C++**
Add a new subclass	Potential set of subclasses is unbounded	Y	Y	Y	Y	Y	Y
Define an overriding method in subclasses	Static type information and other information of the form "*object is of class C or one of its subclasses*" is useless for static-binding of message sends (since a new subclass with an overriding definition of the method could be defined in another module)	Y	Y	Y	Y	Y	Y
Add a method to an existing class	Cannot perform compile-time method lookup past any point where a method is *not* defined, since a definition of that method could appear in another module	Y	Y	Y			
Add a parent class to an existing class	Cannot perform compile-time method lookup across any inheritance links, since new parents (and hence inheritance paths) might be defined in other modules.	Y					
Add a method with a new dispatched argument position to a generic function	Must assume that any argument positions of a message might influence method lookup. Method dispatching must assume the worst case.	Y	Y				
Predicate methods [Chambers 93b]	Cannot perform compile-time method lookup *at all*, since a predicate method might be defined in another modules that overrides any visible method.	Y					

insufficient to guarantee that the whole program will be available for optimization). Recent examples of lin-time optimizers include ATOM [Srivastava & Eustace 94] and EEL [Larus & Schnarr 95. The difficulties with link-time optimizations are two-fold. First, there is often insufficient semantic information available in the object code to reconstruct the high level information necessary to perform optimizations like static class analysis. Second, moving optimizations into the linker requires that they be performed each time that the program is linked, and the link step can become a serious impediment for fast turnaround time after programming changes (although the techniques described in Chapter 5 could be applied to an incremental linker to avoid reperforming optimizations that are unaffected by the new or changed object modules being linked). Instead of performing optimizations at link-time, this thesis advocates the use of whole-program analysis and optimization in the compiler. In such a system, the compiler is given access to the source code of the entire program, but still follows the model of generating object files with relocatable machine code in them for each source module. This is illustrated in Figure 2.9: the compiler sees all the source code for the entire

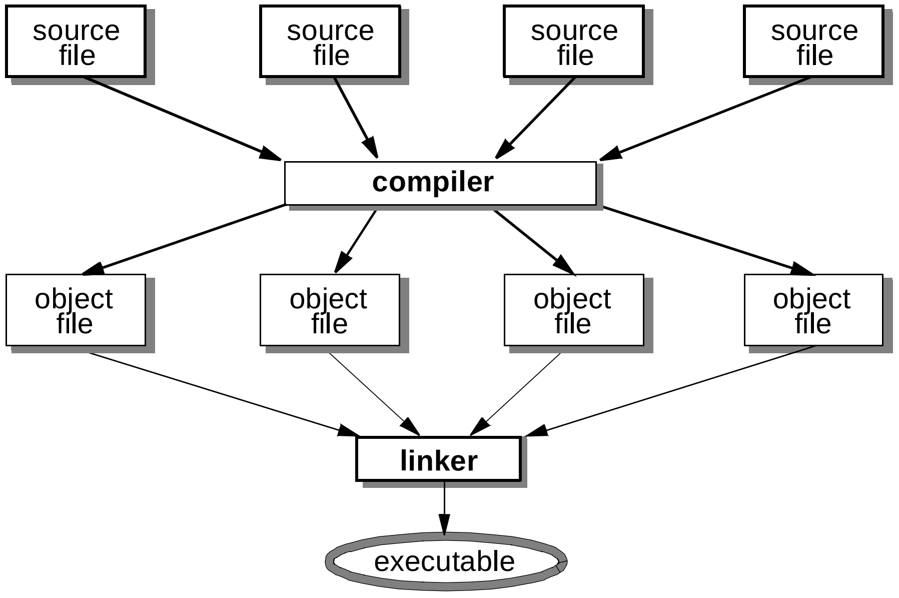


Figure 2.9: Whole program compilation model

program and can use information about its structure when generating each of the object files for the program. Object files still have a correspondence with source files, but cross-module optimizations may have been applied, making an object module potentially dependent not just on its source file but also on various aspects of the other source files in the program. Chapter 5 discusses ways to support incremental recompilation in the presence of cross-module optimizations. Many integrated programming environments such as Borland's Delphi Pascal system and Microsoft's Visual C++ environment already keep track of the source code for the whole program to support tools such as class browsers, so giving the optimizer access to the whole source code in such systems may not require radical changes.

2.5 The Vortex Compiler

To support experimentation with whole-program optimization, we have built the Vortex compiler, an optimizing compiler for object-oriented languages, which provides a useful experimental infrastructure for several reasons. First, it is the first compiler that is able to apply modern optimizations such as profile-guided receiver class prediction and class hierarchy analysis to multiple languages. We currently have front-ends for Cecil [Chambers 93a], a

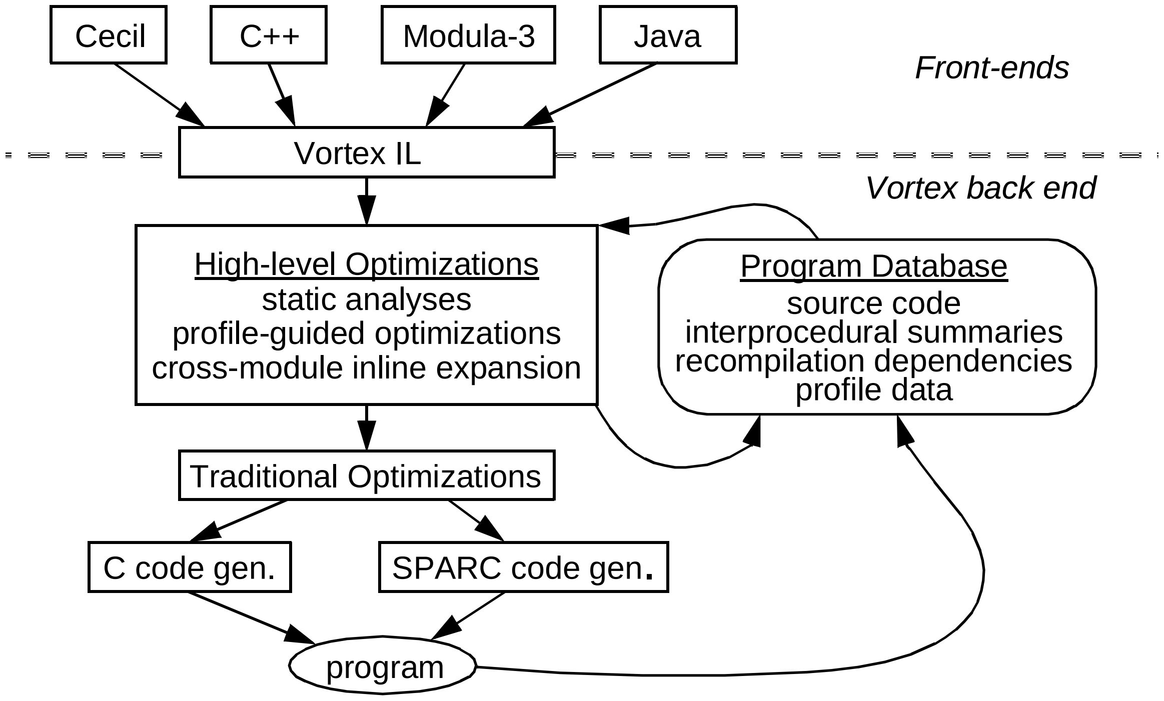


Figure 2.10: Vortex Compiler Architecture

dynamically-typed purely object-oriented language, C++ [Stroustrup 91] and Modula-3 [Nelson 91], two statically-typed hybrid object-oriented languages, and Java [Gosling et al. 96], a mostly statically-typed, mostly purely object-oriented language. Second, the integration of all of these techniques in the common back-end ensures that they are applied in the same, consistent manner across the different languages.

The basic structure of the compiler is shown in Figure 2.10. Each of the different front-ends does whatever parsing and typechecking are appropriate for its input language, and then translates the input into the Vortex compiler's intermediate language (IL). Details of this intermediate language can be found elsewhere [Dean et al. 96]. A suite of high-level optimizations is then applied, aimed at eliminating dynamic dispatches and optimizing away operations such as dead or partially dead closure and object creations. High level operations in the IL are then expanded into a series of lower-level operations, revealing additional opportunities for traditional optimizations such as common subexpression elimination. The compiler can then either generate portable C code or it can further lower the IL, perform low-level optimizations such as register allocation and instruction scheduling, and generate SPARC assembly code.

2.6 Summary

Message sends are expensive and are becoming more so on modern machines. Many optimization techniques, including static analyses and class prediction, can reduce the cost of dynamic dispatches, but these techniques are most effective when the compiler has access to the whole program, using a compilation model similar to that employed by the Vortex compiler. The remainder of this thesis focuses on some of the optimization techniques enabled by whole program optimization and on issues related to making it practical for everyday use in integrated programming environments.

Chapter 3

Exploiting the Class Hierarchy

A serious impediment to optimizing message sends in compilers is that worst-case assumptions must be made about the presence of additional subclasses and overriding definitions of methods in these subclasses. However, these worst-case assumptions can be avoided if the compiler is given knowledge of the whole program's class hierarchy and method declarations. By exploiting information about the structure of the class inheritance graph, including where methods are defined, the compiler can gain valuable static information about the possible classes of the receiver of each method being compiled.

To illustrate this, consider the class hierarchy shown in Figure 3.7, where the method *p* in the class *F* (denote this *F::p*) contains a send of the *m* message to self. *m* is declared to be a virtual function, and there are several implementations of *m* for subclasses of *A*. As a result, with only static intraprocedural class analysis, the send of the *m* message in *F::p* must be implemented as a general message send. However, by examining the subclasses of *F* and determining that there are no overriding implementations of *m*, the *m* message can be replaced with a direct procedure call to *C::m* and then further optimized with inlining, interprocedural analysis, and any other optimizations that can be applied to standard procedure calls. This reasoning depends not on knowing the exact class of the receiver, as with most previous techniques, but rather on knowing that no subclasses of *F* override the version of *m* inherited by *F*. Some languages, including C++ and Java, allow the programmer to explicitly provide the compiler with information that a particular method will not be overridden, through explicit source-level annotations on methods. These source-level approaches and their drawbacks are discussed in Section 3.1. This chapter focuses on automatic compiler optimizations that can achieve the same or better effect than these programmer annotations, without programmer intervention. Class hierarchy analysis is a simple analysis to statically-bind messages when only a single method can be invoked from a particular call site, and is discussed in Section 3.2.

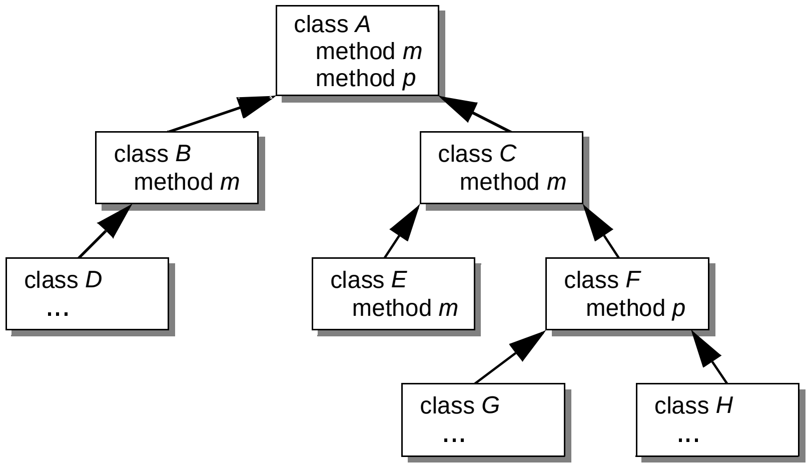


Figure 3.1: Example class hierarchy

Exhaustive testing is a technique that can statically-bind a message send when more than a single method is possible, through the insertion of explicit tests, and is discussed in Section 3.3. The performance of these two techniques is evaluated on a suite of medium-to-large benchmark programs in Section 3.4. Section 3.5 discusses some other issues related to making class hierarchy analysis practical for large programs, and Section 3.6 discusses related work.

3.1 Source-Level Program Annotations

Many languages include source-level mechanisms for achieving a similar effect to class hierarchy analysis. For example, in C++, programmers can declare whether or not a method is virtual (methods default to being non-virtual) [Stroustrup 91]. When a method is not declared to be virtual, the compiler can infer that no subclass will override the method,[1] thus enabling it to implement invocations of the method as direct procedure calls. However, this approach suffers from three weaknesses relative to class hierarchy analysis:

1. Actually, C++ non-virtual functions can be overridden, but dynamic binding will not be performed: the static type of the receiver determines which version of the non-virtual method to invoke, not the dynamic class. This behavior is not usually desired and can be the source of subtle bugs.

- The C++ programmer must explicitly decide which methods need to be virtual, making the programming process more difficult. When developing a reusable framework, the framework designer must decide which operations will be overridable by clients of the framework, and which will not. The decisions made by the framework designer may not match the needs of the client program. In particular, a well-written, highly-extensible framework will often provide flexibility that is used by some applications but not by others. In contrast, class hierarchy analysis is automatic and adapts to the particular framework/client combination being optimized.
- The virtual/non-virtual annotations are embedded in the source program. If extensions to the class hierarchy are made that require a non-virtual function to become overloaded and dynamically dispatched, the source program must be modified. This can be particularly difficult in the presence of separately-developed frameworks that clients may not be able to change. Since class hierarchy analysis is an automatic mechanism, it requires no source-level modifications.
- A function may need to be virtual, because it has multiple implementations that need to be selected dynamically, but within some particular subtree of the inheritance graph, there may be only one implementation that applies. In the example above, the *m* method must be declared virtual, since there are several implementations, but there is only one version of *m* that is called from *F* or any of its subclasses. In C++, *m* must be virtual and consequently the send would be implemented with a dynamically-bound message. In contrast, class hierarchy analysis can identify when a virtual function "reverts" to a non-virtual one with a single implementation for a particular class subtree, enabling better optimization.

C++ is not the only language that provides the ability for programmers to control what sort of extensions are to be allowed to a program. Java has final methods, which are somewhat more flexible than virtual functions, but still suffer from the same problems discussed above. Another common language feature is the ability to declare a class as a leaf class, exemplified by Trellis's no_subtypes annotation [Schaffert et al. 85, Schaffert et al. 86], Dylan's seal annotation [Dyl92], and Java's final annotation when applied to a class declaration [Gosling et al. 96]. All of these are ways to inform the compiler that there will be no subclasses of the class, which can optimization of messages sent to an isntance of the class. In most object-oriented languages, mesage sends to an instance of a leaf class can always be optimized, since no additional method declarations can be present that affect that class other than the declarations that are already visible to the compiler. Some languages, such as Cecil and Dylan, permit new methods to be defined externally from the class definition. In these languages, the mere knowledge that a class is a leaf class does not permit optimization of messages sent to it. Declaring a class as a leaf class has similar problems to declaring a method as non-virtual: programmers must make this declaration explicitly, as part of the source code, and must anticipate what kinds of future extension are going to be required in the software. In contrast, class hierarchy analysis discovers more precise information than any of these annotations and does so automatically.

3.2 Implementation of Class Hierarchy Analysis

To make class hierarchy analysis effective, it must be integrated with intraprocedural static class analysis. Static class analysis is a kind of data flow analysis that computes a set of classes for each variable and expression in a method, which the compiler uses to optimize dynamically-bound messages, type-case statements as in Modula-3 and Trellis, and other run-time type checks. Previous frameworks for static class analysis in dynamically-typed object-oriented languages have defined several representations for sets of classes [Chambers & Ungar 90]. Typical representations are summarized in Table 3.1.

Table 3.1: Representations for static class information

Representation	**Description**	**Source**	**Use**
Unknown	the set of all classes	method arguments; results of non-inlined message sends; contents of instance variables	
Class(C)	the singleton set $\{C\}$	true branch of run-time class tests; literals	supports static binding of sends; eliminating run-time type checks
Union(S_1, ..., S_n)	union of class sets	control flow merges	supports “type-casing” if small union of classes
Difference(S_1, S_2)	difference of two class sets	false branch of run-time class tests	avoids repeated tests

Earlier class analysis frameworks focused on the singleton class set as the primary source of optimization: if the receiver of a message is a singleton class set, then the message lookup can be resolved at compile-time and replaced with a direct procedure call to the target method. Unions of class sets were optimized only through a type-casing optimization, if the union combined a small number of classes.

3.2.1 Cone Class Sets

Class hierarchy analysis changes the flavor of static class analysis. The initial class set associated with the receiver of the method being analyzed is the set of classes inheriting from the class containing the method; in the earlier example, the receiver of *F*'s *p* method is associated with the set {*F*, *G*, *H*}. It would be possible to use the Union set representation to represent the class set of the method receiver, but this could be space-inefficient for the large receiver class sets of methods declared high up in the inheritance hierarchy. Consequently we introduce a new representation for the kind of regular class sets inferred by class hierarchy analysis, the Cone, described in Table 3.3. Cone representations are essentially a representation of a union of individual classes that happen to be all the subclasses of a common superclass *C*.

Table 3.2: Cone representation of static class information

Representation	**Description**	**Source**	**Use**
Cone(*C*)	the set of all subclasses of the class *C*, including *C*	class hierarchy analysis of method receiver; static type declarations	supports static binding of sends

Class hierarchy analysis annotates the method's receiver with a cone set representation for the class containing the method. A simple optimization of this representation is to use the Class(*C*) representation rather than Cone(*C*) if *C* is a leaf class or if Cone(*B*) contains only a single concrete (non-abstract) class *C*. Cones tend to be concise summaries of sets of classes: in our implementation, when compiling the Vortex compiler at a point when it was 52,000 lines long and contained 957 classes, the average cone used for optimization purposes contained 12 concrete classes, and some cones included as many as 93 concrete classes.

In a statically-typed language, cones can be used to integrate static type declarations into the static class analysis framework: for a variable declared to be of static type *C*, any static class information inferred for the variable is intersected with Cone(*C*). This provides a simple way of integrating static type information in with static class analysis, enabling class hierarchy analysis to be effectively applied to statically-typed object-oriented languages. For hybrid languages, built-in non-object-oriented data types like integers and arrays can be considered their own separate classes, as far as static class analysis is concerned.

3.2.2 Method Applies-To Sets

If only singleton class sets support static binding of messages, then only leaf classes would benefit from class hierarchy analysis. However, this is unnecessarily conservative: even if the receiver of a message has multiple potential classes, if all the classes inherit the same method, then the message send can be statically bound and replaced with a direct procedure call. For instance, in the earlier example, the class set computed for the *m* message sent to the receiver of the *F*::*p* method is {*F*, *G*, *H*}, but all three classes inherit the same implementation of *m*, *C*::*m*. Measurements from compiling Vortex when it was 52,000 lines long indicate that nearly 50% of the messages statically bound using class hierarchy analysis have receiver class sets containing more than a single class. To receive the most benefit from class hierarchy analysis, static binding of messages whose receivers are sets of classes should be supported. One approach would be to iterate through all elements of Union and Cone sets, performing method lookup for each class, and checking that each class inherits the same method. However, this has the disadvantage of being slow for large sets, such as cones of classes with many subclasses.

We have pursued an alternative approach that compares entire sets of classes at once. We first precompute for each method the set of classes for which that method is the appropriate target, which we call the *applies-to* set. In our compiler, these sets are computed on demand, the first time a message with a particular name and argument count is analyzed, to spread out the cost of this computation. Then at a message send, we take the class set inferred for the receiver and test whether this set overlaps each potentially-invoked method's applies-to set. If only one method's applies-to set overlaps the receiver's class set, then that is the only method that can be invoked and the message send can be replaced with a direct procedure call to that method. To avoid repeatedly checking a large number of methods for applicability at every call site in the program, our compiler incorporates a compile-time method lookup cache that memoizes the function mapping receiver class set to set of target methods. In practice, the size of this cache is reasonable: for a 52,000-line Cecil program, this cache contained 7,686 entries, and a total of 54,211 queries of the cache were made during compilation. This cache also provides a natural point for handling dependencies due to class hierarchy optimizations, as discussed in Chapter 5.

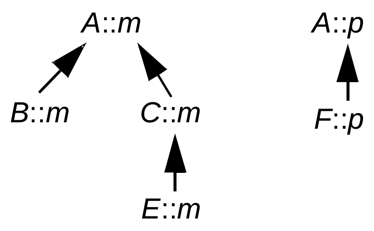


Figure 3.2: Method partial orders for running example

The efficiency of this approach depends on the ability to precompute the applies-to sets of each method and the implementation of the set overlaps test for the different representations of sets. To precompute the applies-to sets, we first construct a partial order over the set of methods, where one method M_1 is less than another M_2 in the partial ordering if and only if M_1 overrides M_2. For the running example (introduced in Figure 3.7), we construct the partial order shown in Figure 3.2. Then, for each method defined on class *C*, we initialize its applies-to set to Cone(*C*). Finally, we traverse the partial order top-down. For each method *M*, we visit each of the immediately-overriding methods and subtract off their (initial) applies-to sets from *M*'s applies-to set. In general, the resulting applies-to set for a method *C*::*M* is represented as Difference(Cone(*C*), Union(Cone(D_1), ..., Cone(D_n))), where D_1, ..., D_n are the classes containing the directly-overriding methods. If a method has many directly-overriding methods, the representation of the method's applies-to set can become quite large. To avoid this problem, the subtraction can be ignored at any point; it is safe though conservative for applies-to sets to be larger than necessary.

The efficiency of overlaps testing depends on the representation of the two sets being compared. Overlaps testing for two arbitrary Union sets of size N takes time $O(N^2)$.[2] However, overlaps testing among Cone and Class representations takes only constant time, assuming that testing whether one class can inherit from another takes only constant time [AK et al. 89, Agrawal et al. 91, Caseau 93]. For example, Cone(C_1) overlaps Class(C_2) if and only if $C_1 = C_2$, or C_2 inherits from C_1 or vice versa. Overlaps testing of arbitrary Difference sets is complex and can be expensive. Since applies-to sets in general are Differences, the cost of testing to see if a receiver class set overlaps with a collection of applies-to Difference sets can be substantial. To represent irregular applies-to sets more efficiently, we convert Difference sets into a flattened BitSet representation, with one bit position to represent each class in the program (a

2. Since the set of classes is fixed, Union sets whose elements are singleton classes can be stored in a sorted order, reducing the overlaps computation to O(N).

BitSet is effectively a specialized representation for large Union class sets). Overlaps testing of two BitSet class sets requires $O(N)$ time, where N is the number of classes in the program. In practice, this check is fast: even for a large program with 1,000 classes, if bit sets use 32 bit positions per machine word, only 31 machine word comparisons are required to check whether two bit sets overlap. In our implementation, we precompute the BitSet representation of Cone(C) for each class C, and we use these bit sets when computing differences of Cones, overlaps of Cones, and membership of a class in a Cone.

When compiling a method and performing intraprocedural static class analysis, the static class information for the method's receiver is initialized to Cone(C), where C is the class containing the method. It might appear that the applies-to set computed for the method would be more precise initial information. Normally, this would be the case. However, if an overriding method contains a super send (or the equivalent) to invoke the overridden method, the overridden method can be invoked with objects other than those in the method's applies-to set; the applies-to set is only correct for normal dynamically-dispatched message sends, not for super sends. If it is known that none of the overriding methods contain super sends that would invoke the method, then applies-to would be a more precise and legal initial class set.

3.2.3 Support for Multi-Methods

The strategy for static class analysis in the presence of class hierarchy information described in Section 3.2.1 and Section 3.2.2 works for singly-dispatched languages (where methods dispatch on only a single argument position), but it does not support languages with multi-methods (where any or all of the arguments to a message send can contribute to determining what method is invoked), such as CLOS, Dylan, and Cecil. This section extends the static analysis framework to handle multi-method based languages. To support multi-methods, we associate methods not with sets of classes but sets of k-tuples of classes, where k is the number of dispatched arguments of the method.[3] To represent many common sets of tuples of

3. We assume that the compiler can determine statically which subset of a message's arguments need to be examined as part of method lookup. In CLOS, for instance, all methods in a generic function have the same set of dispatched arguments. In Cecil, the Vortex compiler exploits whole-program information by examining all methods with the same name and number of arguments to find all argument positions that any of the methods is specialized upon. It would be possible to consider all arguments as potentially-dispatched, but this would be substantially less efficient, both at compile-time and at run-time, particularly if the majority of methods are specialized on only a single argument.

classes concisely, we use *k*-tuples of class sets: a *k*-tuple $<S_1, ..., S_k>$, where the S_i are class sets, represents the set of tuples of single classes that is the cartesian product of the S_i class sets. To represent other irregular sets of tuples of classes, we support a union of class set tuples as a basic representation.

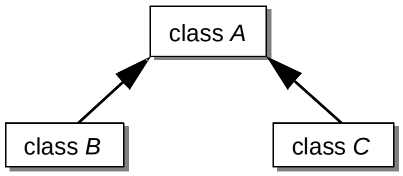
Static class analysis in the presence of multi-methods is similar to the singly-dispatched case, with the following differences. For each method, we compute the method's applies-to *tuple* of class sets, which describes the combinations of classes for which the method should be invoked. For a multi-method specialized on the classes C_1, ..., C_k, the method's applies-to tuple is initialized to <Cone(C_1), ..., Cone(C_k)>. When visiting the directly-overriding methods, the overriding method's applies-to tuple is subtracted from the overridden method's tuple. When determining which methods apply to a given message, the *k*-tuple is formed from the class sets inferred for the *k* dispatched message arguments, and then the applies-to tuples of the candidate methods are checked to see if they overlap the tuple representing the actual arguments to the message.

Efficient multi-method static class analysis relies on efficient overlaps testing and difference operations on tuples. Testing whether one tuple overlaps another is straightforward: each element class set of one tuple must overlap the corresponding class set of the other tuple. Computing the difference of two tuples of class sets efficiently is trickier. The pointwise difference of the element class sets, though concise, would not be a correct implementation. One correct representation would be a union of *k* *k*-tuples, where each tuple has one element class set difference taken:

$$<S_1, ..., S_k> - <T_1, ..., T_k> \equiv \bigcup_{i=1..k} <S_1, ..., S_{i-1}, S_i - T_i, S_{i+1}, ..., S_k>$$

If the $S_i - T_i$ element set is empty, then the *i*-th *k*-tuple is dropped from the union, since its cartesian-product expansion is the empty set. Also, if two tuples in the union are identical except for one position, they can be merged into a single tuple by taking the union of the element class sets. Both of these optimizations are important in practice to reduce the number of terms explicitly represented.

For example, consider the class hierarchy and multi-methods shown in Figure 3.3 (`x@`*X* is the syntax we use for indicating that the `x` formal argument of a multi-method is specialized for the class *X*).



```
m(r@A, s@A, t@A) { ... }

m(r@B, s@C, t@A) { ... }
```

Figure 3.3: Example class hierarchy and multi-methods

Under both CLOS's and Cecil's method overriding rules, the partial order constructed for these methods is shown in the following:

$$\begin{tikzcd}
\texttt{m(@}A\texttt{, @}A\texttt{, @}A\texttt{)} \\
\texttt{m(@}B\texttt{, @}C\texttt{, @}A\texttt{)} \arrow[u]
\end{tikzcd}$$

Figure 3.4: Multi-method partial order

The applies-to tuples constructed for these methods, using the formula above, are:

`m(@A, @A, @A)`:	$<\{A,C\}, \{A,B,C\}, \{A,B,C\}> \cup <\{A,B,C\}, \{A,B\}, \{A,B,C\}>$
`m(@B, @C, @A)`:	$<\{B\}, \{C\}, \{A,B,C\}>$

(The third tuple of the first method's applies-to union drops out, since one of the tuple's elements is the empty class set.)

Unfortunately, for a series of difference operations, as occurs when computing the applies-to tuple of a method by subtracting off each of the applies-to tuples of the overriding methods, this representation tends to grow in size exponentially with the number of differences taken. For example, if a third method is added to the existing class hierarchy, which overrides the first method but not the second:

```
m(r@C, s@B, t@C) { ... }
```

then the applies-to tuple of the first method becomes the following:

`m(@A, @A, @A)`: $<\{A\}, \{A,B,C\}, \{A,B,C\}> \cup <\{A,C\}, \{A,C\}, \{A,B,C\}> \cup$
$<\{A,C\}, \{A,B,C\}, \{A,B\}> \cup <\{A,B\}, \{A,B\}, \{A,B,C\}> \cup$
$<\{A,B,C\}, \{A\}, \{A,B,C\}> \cup <\{A,B,C\}, \{A,B\}, \{A,B\}>$

To curb this exponential growth problem, we have developed, with help from William Pugh, a more efficient way to represent the difference of two overlapping tuples of class sets:

$$<S_1, ..., S_k> - <T_1, ..., T_k> \equiv \cup_{i=1..k} <S_1 \cap T_1, ..., S_{i-1} \cap T_{i-1}, S_i - T_i, S_{i+1}, ..., S_k>$$

By taking the intersection of the first i-1 elements of the i-th tuple in the union, we avoid duplication among the element tuples of the union. As a result, the element sets of the tuples are smaller and tend to drop out more often for a series of tuple difference operations. For the three multi-method example, the applies-to tuple of the first method is simplified to the following:

`m(@A, @A, @A):` $<\{A\}, \{A,B,C\}, \{A,B,C\}> \cup <\{C\}, \{A,C\}, \{A,B,C\}> \cup$
$<\{C\}, \{B\}, \{A,B\}> \cup <\{B\}, \{A,B\}, \{A,B,C\}>$

As a final guard against exponential growth, we impose a limit on the number of class set terms in the resulting tuple representation, beyond which we stop narrowing a method's applies-to set. We rarely resort to this conservative approximation: when compiling the Vortex compiler when it was a 52,000-line Cecil program, only one applies-to tuple, for a message with 5 dispatched argument positions, crossed our implementation's threshold of 64 terms. The intersection-based representation is crucial for conserving space: without it, many applies-to sets would have exceeded the 64-term threshold.

3.2.4 Support for Dynamically-Typed Languages

In a dynamically-typed language, it is possible that for some receiver classes a message send will result in a run-time message-not-understood error. When attempting to optimize a dynamic dispatch, we need to ensure that we will not replace such a message send with a statically-bound call even if there is only one applicable source method. To handle this, we introduce a special "error" method defined on the class(es) at the top of the class hierarchy, if there is no default method already defined. Once error methods are introduced, no special efforts need be made to handle the possibility of run-time method lookup errors. For example, if only one source method is applicable, but a method lookup error is possible, our framework will consider this case as if two methods, one real and one error, were applicable and hence block static binding to the one real method. Similarly, if a message is ambiguously defined for some class, more than one method will include the class in its applies-to set, again preventing static binding to either method.

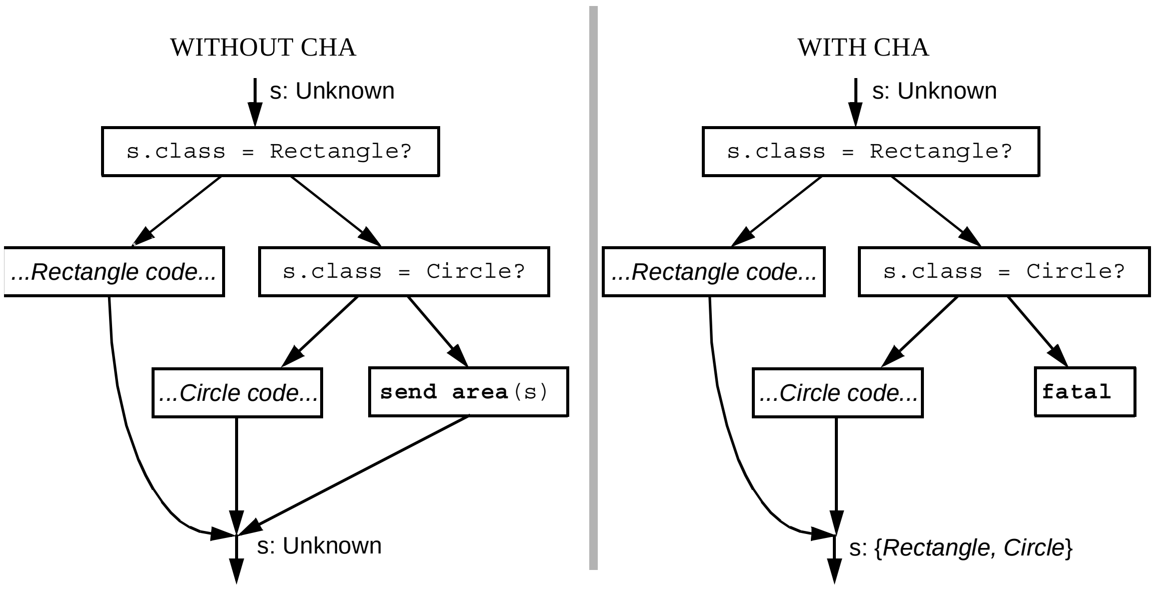


Figure 3.5: Class prediction with and without class hierarchy analysis

3.2.5 Improving Class Prediction

Class prediction, discussed in Section 2.3.2, is a common technique to optimize message sends, by inserting explicit tests for likely classes at a particular call site. Without knowledge of the complete hierarchy, the compiler must insert a full message send in the else case after a sequence of class tests, to handle the case where the class is not one of the predicted-for classes. However, given knowledge of the complete class hierarchy, the compiler can sometimes optimize the else branch, if it can determine that a particular sequence of tests has exhaustively covered all the cases. In dynamically-typed languages, if the compiler can prove statically that the classes being tested exhaust the set of classes for which the message is correctly defined, then the final “unexpected” case can be replaced with a run-time message lookup error trap. Such a lookup error trap might take up less compiled code space than a full message send, but more importantly in some languages it is known not to return to the caller. Thus, the error branch never merges back into the main stream of the program, and the compiler learns that only the predicted class(es) are possible after the message. For example, consider the control flow graph shown in Figure 3.5. On the left is the code for class prediction without class hierarchy analysis, where the else case is implemented as a full message send of the area message. In the above example, if analysis of the class hierarchy reveals that Rectangle and Circle are the only classes implementing the area message, then the third case can be replaced with an error trap. After the area message, the compiler will know that s is either a

Rectangle or a Circle, enabling it to better implement later messages sent to s. (In a statically-typed language, class hierarchy analysis coupled with static type declarations would have shown s to refer to either a Rectangle or a Circle all along, leading to the insertion of only a single test to distinguish the two cases.) In the absence of class hierarchy information, the compiler must assume that some other class could implement the area message, and consequently include support for the third "unexpected" case. When compiling the Vortex compiler when it was an 80,000-line Cecil program, elimination of unexpected cases using class hierarchy analysis occurred 9,402 times; 3,603 of these occurrences optimized basic messages such as if and not, which might not be necessary in a less-pure language lacking user-defined control structures.

There are other ways of preventing the uncommon cases from polluting the more precise information obtained from a sequence of class tests. Using dynamic compilation, as discussed in Section 2.3.4, the uncommon branches can be compiled lazily [Chambers & Ungar 91, Hölzle 94]. In this approach, instead of compiling code for the uncommon case, a simple stub is inserted to invoke the compiler and generate the appropriate code when and if the uncommon case happens. This allows downstream code to be compiled as if only the common cases will occur. It does require, however, that the compiler be able to reconstruct the program's state when an uncommon branch does occur, and so it places some limitations on what kinds of optimizations can be performed. It also requires an infrastructure that supports dynamic compilation.

Eager splitting of uncommon branches is another approach that avoids merging the uncommon branches back into the control flow graph [Chambers 92]. Whenever an uncommon branch is about to merge back into the common part of the control flow, the compiler prevents the merge from happening by duplicating all the code downstream of the potential merge, and compiling the now-replicated downstream portions of the control flow graph independently (usually compiling the uncommon portion without substantial optimizations, since it is expected to be uncommon). Although this does have the effect of preventing pessimistic assumptions about the behavior of the uncommon cases to cloud the optimization of the common cases, it can come at the expense of (potentially exponential) code explosion, and its application usually has to be tempered with heuristics to prevent this code explosion from becoming unmanageable.

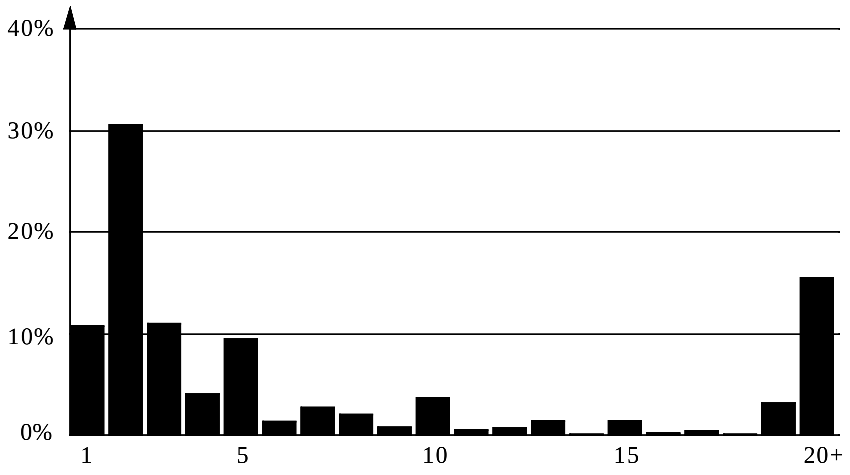


Figure 3.6: Number of candidate methods per static call site

3.3 Exhaustive, Class-Hierarchy-Guided Testing

The previous section discussed the basic mechanism for representing class hierarchy information, for computing applies-to sets for each method, and for using this information to statically-bind message sends when only a single method could be invoked. However, class hierarchy analysis is often not able to narrow the set of potentially-invoked methods at a particular call site to a singleton set, even though it is often able to narrow the set of candidate methods to a small set of potentially invocable methods. As an example of programs where class hierarchy analysis alone is insufficient to optimize many of the message sends, the histogram in Figure 3.6 shows the distribution of the number of potentially-invoked methods that were identified using complete class hierarchy information at the 52,891 message send sites that were analyzed during the compilation of the Vortex compiler when it was a 75,000-line Cecil program. For the purposes of the histogram, since Cecil is dynamically-typed, the possibility of a message-not-understood error was treated as an additional candidate method.

As the histogram illustrates, only 11% of the static call sites have a single candidate method and are able to be statically-bound using class hierarchy analysis alone. However, 43% of the call sites have only 2 or 3 candidate methods. In this section, we explore the possibility of inserting class predictions that exhaustively test for all possibilities at a call site, for those call sites that had more than one applicable method after class hierarchy analysis. Unlike previous approaches to class prediction, this testing is guided by examining the class hierarchy, rather than being driven by profile information or a set of hard-wired heuristics. Section 3.3.1 and Section 3.3.2 consider two mechanisms for supporting class tests, and Section 3.3.3 describes an algorithm that relies on these mechanisms to insert exhaustive tests to statically-bind a send.

3.3.1 Single Class Tests

All object-oriented language implementations are able to compute some unique identifier of the class of an object[4] when a message is sent to the object, since they must be able to determine what method should be invoked based on the runtime class of the object. For example, a virtual function table pointer in C++, a method suite pointer in Modula-3, a class object in Smalltalk, and a map identifier in Self and Cecil all serve the purpose of uniquely identifying an object's class. The cost of testing for a single class is typically only a few instructions on most modern machines, usually involving loading the identifier out of the object, comparing it against a compile-time constant, and branching. On a SPARC [Sun91], for example, it can be done in 3 or 5 instructions, depending on whether or not the compile-time constant fits in the immediate field of the comparison. This makes it feasible to insert several tests to partition the possible classes of the receiver of a message send into the different sets that invoke different methods. For example, consider the class hierarchy and code fragment in Figure 3.7. Class hierarchy analysis will be able to narrow down the possible methods invoked by the send m.obj to three candidate methods (method A::m, method B::m, and method C::m), but this information alone is insufficient to statically-bind the call site to a single method. Through the insertion of explicit tests, however, the call site can be transformed to three different cases, each of which invokes a single method and therefore is statically-bound. In many respects, the compiler transforms the code in much the same way that a programmer might manually write the code using a TYPECASE statement in Modula-3, for example. One possible sequence of tests is shown in Figure 3.7.

There are two main benefits to exhaustive testing:

4. The implementations of classless languages such as Self and Cecil introduce the notion of the *map* of an object at the implementation level as a means of grouping together objects with the same inheritance characteristics [Chambers 92]. Maps serve the same purpose in the implementation as class identifiers in class-based languages.

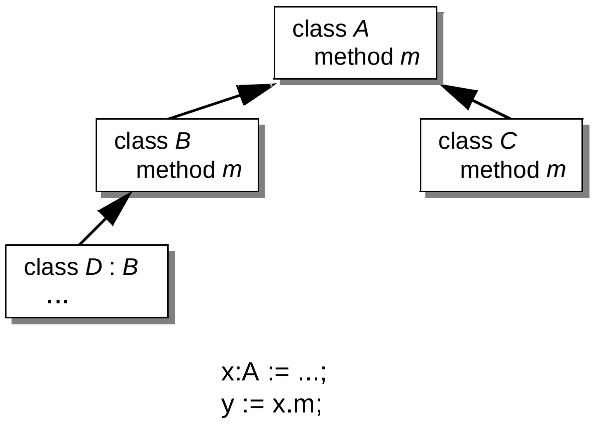


Figure 3.7: Example class hierarchy and code for class tests

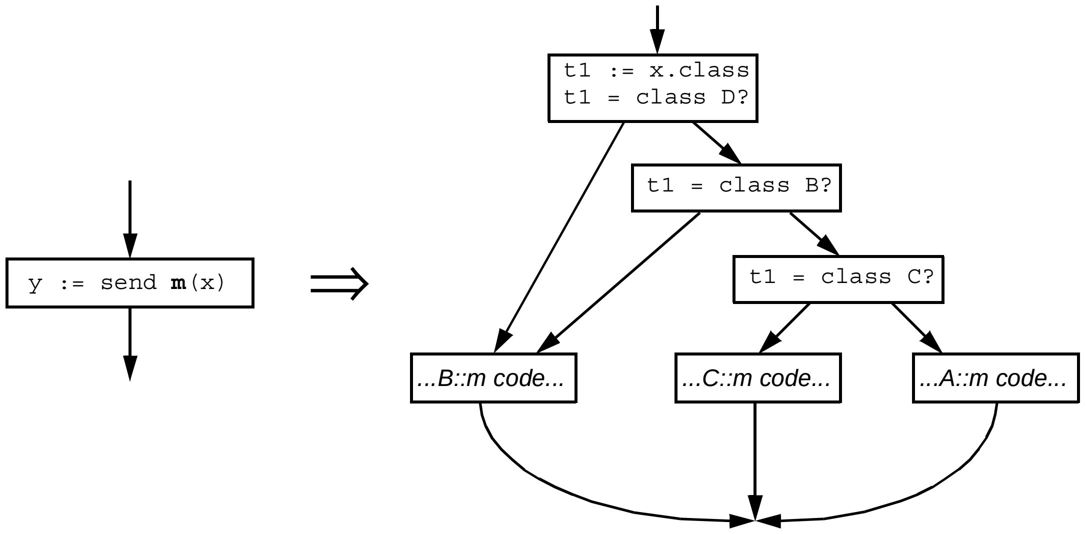


Figure 3.8: Static binding through exhaustive tests

- *All paths become statically-bound*. All possible cases for the message send are resolved to statically-bound calls in the callee, which are then amenable to inlining.
- *Downstream information improves*. If all applicable methods end up being inlined, the optimizer can have more precise information about the result of the message send, resulting in beneficial downstream effects, and it can obtain these downstream benefits while performing only *intra*procedural dataflow analyses.

Despite these advantages, there are some pitfalls in using tests for single classes as a means of exhaustively testing:

- *Testing for large hierarchies is impractical.* Because each test can test for only a single class at a time, using exhaustive testing is impractical at call sites where a large number of classes but only a few methods are possible. For large hierarchies, single class tests suffer both in terms of code space and in terms of execution speed. For example, if class B, instead of having just a single subclass D, had 50 different subclasses, then the exhaustive testing performed in Figure 3.7 would be impractical because it would require too many tests. Not only such testing cause a large blowup in compiled code space but, assuming that the compiler had no information about which classes were most likely, would probably slow the program down by executing many unsuccessful tests before reaching a successful test.

- *Increased sensitivity to programming changes is possible.* Exhaustive testing using singleton class tests increases the possibility that source changes will require recompilation of the generated code for a message send. In particular, adding new subclasses that normally would not affect the set of candidate methods now impacts the correctness of the code because the possible subclasses are exhaustively enumerated in the tests. For example, if a new subclass E is defined that inherits from class B, then the exhaustive testing done in Figure 3.7 would no longer be correct for the call site and it would need to be recompiled.

  This increased sensitivity to new subclasses arises because it is assumed that after testing for *n* - 1 of the cases, the remaining case must be the one that has occurred. This could be avoided if explicit tests are inserted for all *n* branches, and, as with profile-guided class prediction (discussed in Section 2.3.2), an else branch with a full message send is preserved for cases that do not satisfy the tests inserted. However, this eliminates any downstream benefits of exhaustive testing, since there is now always a path with a full message send that imposes worst-case assumptions on all the paths when they merge back together.

The next section addresses these drawbacks through the use of an implementation mechanism that permits efficient testing of subclassing relationships at run-time.

```
N: the current class in the inheritance tree
id: the current identifier value to assign

assign-ids(N, id) =
    N.l := id
    nextID := id + 1
    foreach C ∈ children(N) do
        nextID := assign-ids(C, nextID)
    end
    N.h := nextID
    return nextID + 1

assign-ids(rootClass, 0)        -- Start the id assignment process with the top class in hierarchy
```

Figure 3.9: Assigning class identifiers under single inheritance

3.3.2 Cone Tests

If a language's implementation can provide an efficient single test that answers the question "*does A inherit from class B*?", then both of the drawbacks discussed in Section 3.3.1 can be mitigated, and exhaustive testing can be practically applied to a much larger fraction of call sites. For example, consider the example hierarchy in Figure 3.7, but suppose that both classes *A* and *B* each have an additional 20 subclasses. In this case, exhaustive testing using single class tests is wildly impractical, requiring at least 22 such tests. Using just two **inheritsFrom** tests, however, the call site can be partitioned into the three groups that invoke the three different implementations of method *m*. We term such tests "cone tests", because they test whether an object is included in the set of classes described by the Cone representation of Section 3.2.

The implementation of such a test depends in large part on whether or not the language supports multiple inheritance. For languages that support only single inheritance, such as Modula-3 [Nelson 91], the implementation of a subclass test can take advantage of the fact that the inheritance graph is a tree and not a dag. For trees, it is possible to assign each node a pair of numbers, *l* and *h*, so that, for each node in the tree, the following property holds:

"x is a subclass of y" **iff** $x.l \geq y.l$ and $x.h \leq y.h$

Figure 3.9 presents a simple algorithm to assign *l* and *h* numbers consistent with this property, by walking the tree assigning *l* values on the way down towards the leaves and assigning *h* values on the way back up toward the root.

```
id := obj.class_id;
num := id.l;
if (num >= B.l and num <= B.h) then ...
    ... code for case where obj inherits from B ...
else
    ... obj does not inherit from B ...
```

Figure 3.10: **inheritsFrom(obj,B)** implementation with single inheritance

Once the class hierarchy has been assigned these numbers, then a subclass test of the form “obj **inheritsFrom** class B” can be implemented as a pair of comparisons, shown in Figure 3.12.

For languages that support multiple inheritance, it is not possible to consistently assign l and h values that satisfy the above property, since a node may have multiple parents. To support efficient subclass testing in the presence of multiple inheritance, the Vortex compiler’s runtime system computes an $N \times N$ boolean matrix of subclass relations at program startup time, where N is the number of classes in the program. Each class is assigned a unique number between 0 and N-1, and the $<i,j>$th entry in the matrix indicates whether or not the class whose number is i is a subclass of the class whose number is j. The test can be implemented efficiently by storing a pointer from each class identifier structure to its row of the matrix.[5] Since the class at the top of the cone that we are testing is a compile-time constant, the column number of the matrix (its class identifier) is known statically. When the relationship

5. It can be accessed more efficiently if each row of the matrix is stored at the end of the class data structure, since its offset is then fixed, given the address of the class structure, thus saving an additional load instruction. However, if class data structures are allocated statically this requires knowing the size of each row of the matrix at compile-time.

```
id := obj.class_id;
inheritsRow := id.inheritsMatrixRow;
byteOffset := B.class_number >> 3;-- Compile-time constant
bitOffset := B.class_number & 0x7;-- Compile-time constant
bitMask := 1 << bitOffset;          -- Compile-time constant
if (inheritsRow[byteOffset] & bitMask != 0) then ...
    ... code for case where obj inherits from B ...
else
    ... obj does not inherit from B ...
```

Figure 3.11: **inheritsFrom(obj,B)** implementation with multiple inheritance

matrix is represented as a bit matrix, the code in Figure 3.12 can be used to test the inheritance relationship.

Encoding inheritance relationships in this manner allows an **inheritsFrom** relationship to be tested using a five- or six-instruction sequence on most modern processors: two or three `load` operations, followed by an `and`, `cmp` and a `beq` (although there is no instruction-level parallelism among the instructions, since each instruction is dependent on the result of its predecessor). Furthermore, the matrix is fairly compact, since each entry uses only a single bit.[6] For example, a program with 1000 classes requires 1,000,000 bits, or 125,000 bytes, to represent the matrix, when each row is represented as a bit vector. Most programs with a large number of classes also tend to have a large amount of code and data, and so the relative increase in memory requirements is often not substantial. A potentially serious cost of the cone testing code sequence is likely to be the load instructions that access the **inheritsFrom** matrix, since the access pattern of the relationship matrix is likely to exhibit a low degree of locality if cone tests are used extensively and the program is heavily polymorphic. Alternative implementations of testing inheritance relationships are possible: Aït-Kaci et al. provide a useful overview of efficient lattice operations covering a variety of time-space tradeoffs, and these algorithms would be suitable for answering the "*inherits from?*" query in inheritance hierarchies [AK et al. 89].

6. A faster to query, but less compact representation of the **inheritsFrom** relationship can be obtained by using bytes rather than bits to represent the matrix elements (assuming that the underlying hardware has a byte load instruction). This removes the need for the masking instruction in the test sequence, although the instruction removed is a relatively inexpensive ALU op.

Table 3.3 presents a summary of the costs and other drawbacks of single class tests and cone tests, and also compares it to various common implementations of message dispatching.

Table 3.3: Class testing alternatives

Kind of test	Cost of Dispatch with *N* Alternatives	Drawbacks
Singleton class test	1 load + (1-3)N ALU + N branch	Only tests single class
Cone test (single inheritance)	2 loads + 2N ALU + 2N branches	Only applicable for single inheritance
Cone test (multiple inheritance)	1-2 + N loads + 2N ALU + N branches	Space: Subclass relationship matrix requires $O(N^2)$ space
Table-based dispatch (single inheritance)	2 loads + 1 indirect call	Prevent inlining
Table-based dispatch (multiple inheritance)	3 loads + 1 ALU + 1 indirect call	
Inline caching	1 load + 1-3 ALU + 1 branch + 1 call (hit) … + hash table lookup (miss)	Prevent inlining. Only fast for monomorphic call sites
Polymorphic inline cache	1 load + (1-3)N ALU + N branches + 1 call + 1 jump (hit) … + hash table lookup (miss)	Prevent inlining. Tests one class at time, so can be slow for heavily polymorphic call sites

The presence of a mechanism for doing cone testing greatly increases the applicability of exhaustive testing, since it can readily be applied to large hierarchies. For languages that support multiple inheritance, however, this requires a fair amount of space for the subclass relationship matrix, and the queries of the matrix might have poor cache behavior, since only a single bit or byte of a whole cache line is used to answer each query. A hybrid approach could use the potentially faster single-inheritance cone test when testing in regions of an inheritance hierarchy that were free of multiple inheritance, although this makes the generated code dependent on the fact that a particular subclass hierarchy uses only single inheritance. Another application of cone tests is in the generation of PIC stub routines: as is the case with inline tests, a single cone tests can replace a sequence of singleton class tests. The Vortex compiler's runtime system has recently started applying this technique for its PIC code generation.

3.3.3 Exhaustive Testing Algorithm

The desire to insert exhaustive tests to garner the performance improvements of static binding and inlining must be balanced with the desire to not increase code space and compile time substantially. This section describes a set of heuristics that attempt to balance these concerns in deciding when to insert exhaustive tests, as well as an algorithm for inserting exhaustive tests in an order that allows the use of the **inheritsFrom** test described in Section 3.3.2.

The input to the exhaustive testing decision process is the set of candidate methods that could be invoked from the call site, according to the results of class hierarchy analysis. In order to consider only cases that are likely to improve performance without significantly increasing code space and compile time, our heuristics require the following conditions to be true before inserting exhaustive tests:

1. *The existence of only a few candidate methods.* In order to temper the code space increase, there must be a small number of candidate methods. In our environment we limit this to three or fewer methods, which was chosen to capture a large fraction of the call sites that were not optimized by class hierarchy analysis, but to keep the code space increase manageable.
2. *Most candidates must be inlinable.* The largest performance improvement comes when the statically-bound calls to methods are then inlined and optimized in the context of the call site. Our current heuristics only consider exhaustive testing when at least *n*-1 of the *n* candidate methods are attractive candiates for inlining.

Once a decision is made to insert exhaustive tests, the problem becomes one of choosing the order in which to insert tests. A simple algorithm to generate tests works its way up the method partial order, repeatedly removing one of the bottom methods of the partial order and generating exhaustive tests for this method, until the partial order is empty. The algorithm is shown in Figure 3.12.

This algorithm is designed to make effective use of cone tests by working upwards from the bottoms of the partial order. The methods at the bottom of the partial order can always be tested for using cone tests, and after they have been tested, they can be removed from the partial order. Their removal potentially converts methods that were interior nodes in the partial order into bottoms of the partial order, which can in turn be tested for using cone tests. Although it might sometimes be desirable to test for methods that are interior nodes in the

```
meths: set of candidate methods at call site
info: tuple of static class information known about arguments at call site
specializers(m): tuple of sets of classes that inherit or override this method

exhaustively-test(meths, info) =
  po := build-partial-order(meths)
  while po not empty do
      choose $m \in bottoms(po)$ and remove m from po
      foreach dispatched argument position i do
          if not $info_i$ covers $specializers(m)_i$ then
              insert test(s) to ensure $class(arg_i) \in specializers(m)_i$
          end if
      end for
      -- Along successful test path, it is known that m will be invoked:
      insert statically-bound call to m or inlined version of m, as appropriate

      -- Along unsuccessful test path, we know that classes that invoke m cannot remain
      info := info - specializers(m)
  end while
end
```

Figure 3.12: Algorithm to perform exhaustive testing

partial order first, the “jagged” nature of the applies-to sets for these overridden methods implies that they cannot be tested for using cone tests without first ruling out methods lower in the partial order. Furthermore, a desire to do the testing in this order can only come about when there is an indication that the interior method is more likely to be invoked than the bottom methods. The usual way such an assumption would be formed is through the inspection of profile data. The next section discusses the interaction of profile data with exhaustive testing.

3.3.4 Deciding What Kinds of Tests to Insert

Exhaustive class testing optimizes message sends by creating statically-bound execution paths for all possible classes that might occur at the call site. In contrast, profile-guided class prediction (described in Section 2.3.2) creates optimized execution paths for the commonly-occurring classes and preserves a dynamically-dispatched call site for the (hopefully) uncommon classes. These two different techniques can interact in unfortunate ways. This section first presents an example of how they can interact, and then discusses a set of heuristics to minimize these interactions.

To see how these techniques can interact, consider the class hierarchy and code fragment shown in Figure 3.13. In this hierarchy, a two-argument method, fetch, is defined on the

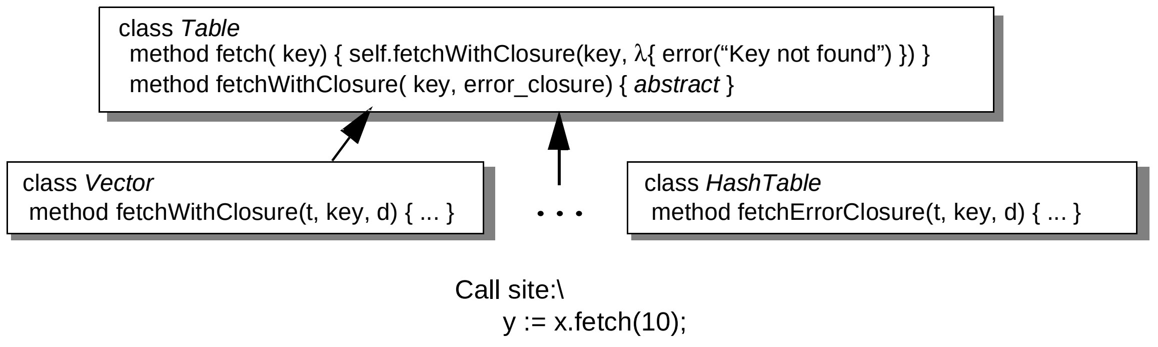
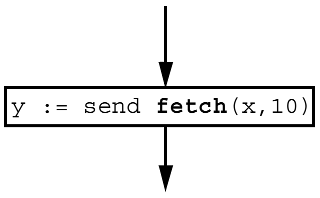

class *Table*
method fetch(key) { self.fetchWithClosure(key, λ{ error("Key not found") }) }
method fetchWithClosure(key, error_closure) { *abstract* }
class *Vector*
method fetchWithClosure(t, key, d) { ... }
• • •
class *HashTable*
method fetchErrorClosure(t, key, d) { ... }
Call site:\
y := x.fetch(10);

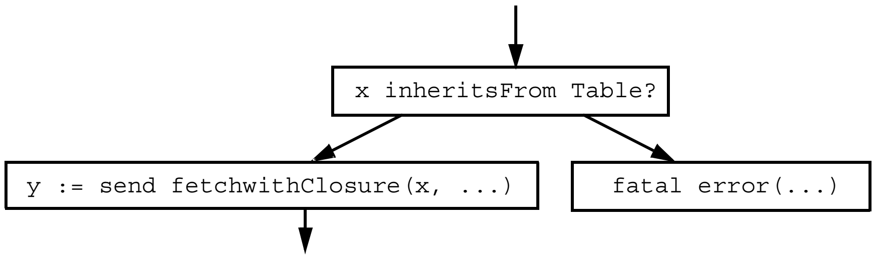
Figure 3.13: Profile and exhaustive testing interaction example

abstract class Table. The various concrete implementations of the Table interface each implement a method for fetching called fetchWithClosure, that takes an additional argument containing a closure to evaluate in case the key is not found in the table. The implementation of fetch in Table simply provides a default implementation of this closure in a send of fetchWithClosure. Making use of factored methods like fetch that then invoke operations defined in subclasses is a common object-oriented programming idiom, as discussed in Section 2.1.

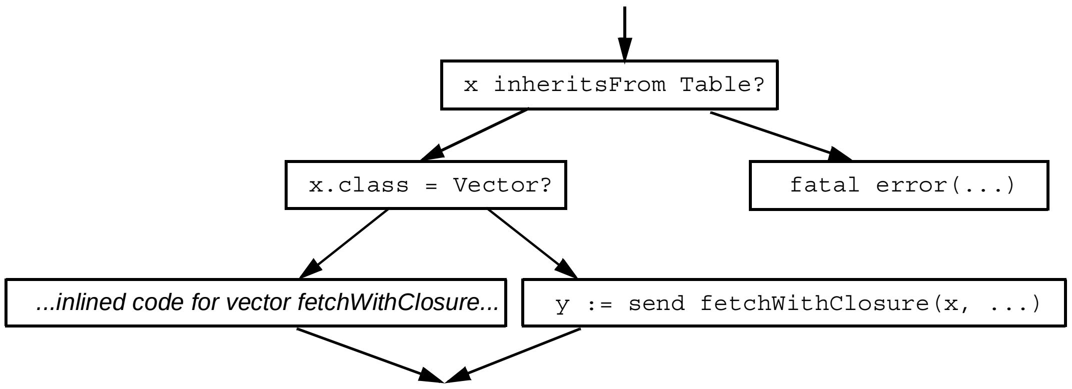
For this example, we will assume that the language is dynamically-typed and that the compiler has no information about the classes of x. In this case, the set of candidate methods at the `x.fetch(10)` call site consists of the version of fetch defined on *Table*, and the implicit msgNotUnderstood method. This call site is an excellent candidate for exhaustive testing and subsequent inlining, giving the code shown in Figure 3.15 (b). After this transformation, the compiler will try and optimize the inlined message send of fetchWithClosure. However, it is only known that x is one of the many classes that descend from Table, and there are enough implementations of fetchWithClosure that exhaustive testing for this call site is impractical. At this point, if profile data is available, a reasonable alternative would be to fall back on inserting profile-guided tests. Figure 3.15 (c) shows the code after inserting a test for Vector, assuming vectors are the most common class at the call site. However, if instead of exhaustively testing at the first message send site for fetch, profile-guided class prediction was used there, this would have resulted in the code shown in Figure 3.15 (d). This code has a fast path where both the send of fetch and the send of fetchWithClosure are statically-bound and inlined using only a single class test, rather than the cone test and class test required by the code in



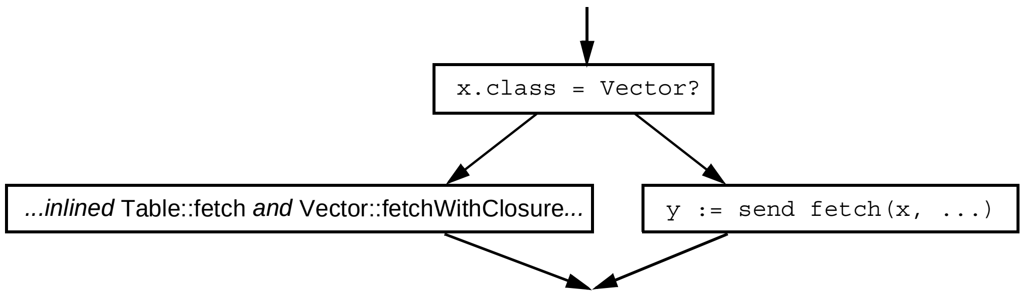
(a) Initial code fragment



(b) After exhaustive testing for x.fetch(10)


x inheritsFrom Table?
x.class = Vector?
fatal error(...)
...inlined code for vector fetchWithClosure...
y := send fetchWithClosure(x, ...)

(c) After profile-guided class prediction for fetchWithClosure message



(d) Using just profile-guided class prediction at call site of x.fetch(10)

Figure 3.14: Profile and exhaustive testing transformations

Figure 3.15 (c). On the other hand, the code in (c) provides faster performance in the presence of instabilities in the commonly-occurring classes at the call site. The following heuristics describe situations in which it is desirable to use exhaustive testing over profile-guided prediction:

- *The profile data is not peaked*. In this case, no one class dominates and so using marginally-slower cone tests that cover all the cases instead of a series of slightly faster class tests is likely to be more than recouped by having a guarantee of exhaustive coverage, rather than having to preserve a full message send on the false path of the profile-guided predictions.
- *The profile data is not stable*. If the compiler can predict that the profile may not be completely representative of all expected inputs to the program, it is probably better to insert exhaustive tests, for the reasons outlined above.
- *There is no downstream need for more precise information*. The problem illustrated in Figure 3.15 arose because a call site downstream of the initial cone test desired more precise information than was garnered by the cone test. This necessitated the insertion of another, more precise test. When there are no downstream places where more precise information is necessary, then there is no disincentive to insert exhaustive but less-precise cone tests. However, detecting downstream desires for more information requires foreseeing how the optimization process will proceed, which is often tantamount to actually doing the downstream optimization. Section 3.3.5 proposes an alternative approach that does not require forseeing the future, whereby initially weak tests are inserted, and, if more precise information turns out to be desirable, these tests are augmented with stronger tests.

Profile information can also guide the exhaustive testing process, by providing likelihood information that can be used to choose a reasonable ordering for inserting the tests. For example, in the ‘choose $m \in$ *bottoms(po)*’ phase of the algorithm, the method in *bottoms(po)* can be selected to be the method that profile information indicates is most likely to be invoked of the remaining candidates.

Determining the best way to optimize a message send, given information about what classes are likely at the call site, requires a careful balancing of the tradeoffs between common-case but non-exhaustive class prediction and exhaustive testing, which is potentially slower in the common case but which can provide additional downstream benefits if all of the potentially invoked methods are tested for and inlined. The core of such an algorithm involves a set of heuristics to help decide which kind of testing strategy to use. The following equations compute the expected cost of using either a sequence of profile-guided class tests, a series of exhaustive class tests, and a combination of the two, given frequency data about the expected distribution of classes at the call site.

mt	cost of a single map test
ct	cost of a single cone test
D	cost of dynamic dispatch
$c_1, c_2, \ldots, c_n$	classes actually appearing at call site
$p_1, p_2, \ldots, p_n$	probabilities of class c_i appearing at call site, $p_{1\check{S}} \geq p_2 \geq \ldots \geq p_n$, and $p_{1\check{S}} + p_2 + \ldots + p_n = 1$

The average cost of a sequence of *i* profile-guided single map tests where the tests are inserted in decreasing order of probability, with a dispatch to handle the remaining *n* - *i* cases, is:

$$MT\ cost = p_1 \times mt + p_2 \times 2mt + \ldots + p_i \times i\ mt + (p_{i+1} + \ldots + p_n)) \times (D + i \times mt)$$

In some cases the static information provided by class hierarchy analysis can show that a series of *i* class tests is exhaustive, as discussed in Section 3.2.4, and so the *i*th test is unnecessary, since it is implied by the failed *i*-1 tests preceding it (i.e. *i* equals *n* in this case, and the compiler is able to determine this). This causes the term for the *i*th test and the dispatch term to drop out.

The cost of using cone tests is slightly more difficult to quantify, since a single cone test may cover multiple possibilities from the distribution of classes. We define:

$M_1, M_2, \ldots, M_m$	*m* methods in applicable method set, in the order for which testing will be done by the algorithm in Section 3.3.3. That is, the methods are ordered according to the method partial order, with ties broken by the frequency data represented by $P(M_i)$
$P(M_i)$	The probability that method M_i will be invoked, computed by summing the p_j for all c_j that invoke M_i

Given these definitions, the cost of using exhaustive cone tests to statically-bind a message send is:

$$ET\ cost = P(M_1) \times ct + P(M_2) \times 2ct + \ldots + P(M_{m-1}) \times (m-1)ct$$

There is no testing for M_m, since the failed tests for the prior *m*-1 methods imply that M_m will be invoked without additional tests.

In some cases, using a combination of class tests and exhaustive cone tests can provide better performance than using either kind of test individually. Since class tests are cheaper than cone tests, this hybrid approach would insert one or a few single class tests first to quickly handle the most common classes, and then insert exhaustive cone tests to handle all the remaining classes. This is probably most useful when the most common classes invoke a method that is high up in the partial order, since the exhaustive testing algorithm for these methods requires that all methods lower in the partial order be tested for first. The cost of this approach, $Hybrid_q$, of quickly testing for the most common q classes (c_1, c_2, …, c_q) using single map tests and then using full exhaustive testing with cone tests, is:

$P_q(M_i)$	Probability that method M_i will be invoked by classes other than $\{c_1, c_2, \ldots, c_q\}$, computed by summing the p_j for all c_j , $j > q$, for which c_j invokes M_i

$$Hybrid_q\ cost = p_1 \times mt + p_2 \times 2mt + \ldots + p_q \times q\ mt + (1 - (p_1 + p_2 + \ldots + p_q))q\ mt + P_q(M_1) \times ct + P_q(M_2) \times 2ct + \ldots + P_q(M_{m-1}) \times (m-1)ct$$

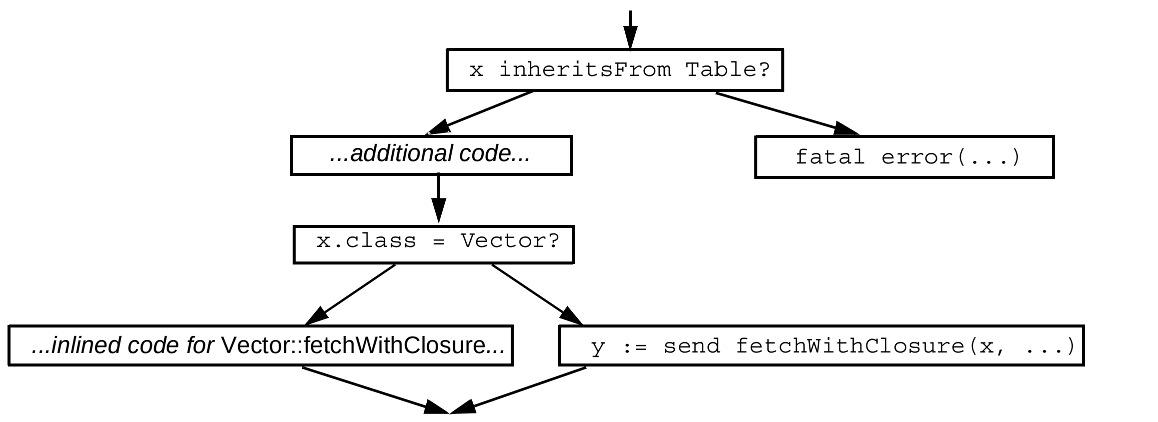
Notice that when q is 0, this expression simplifies to the *ET cost* expression shown above. The performance improvements from the hybrid approach are likely to be fairly small, since exhaustive testing is normally only used for cases where there are a few candidate methods, and so the cost of testing for less common cases before reaching the more common cases is not that severe.

The information needed to compute *MT cost,* and $Hybrid_q$ *cost* (for values of q from 0 to n) is available to the compiler when compiling each message send site, and comparing these values can help to determine whether it is better to insert exhaustive cone tests or profile-guided single class tests. However, it is important to realize that these metrics only compute the cost of implementing the dispatch and do not account for other factors, including exhaustive cone testing's potential for better downstream effects due to the inlining of all paths and better performance in the presence of profile instability, nor for the potential for the more precise information provided by single class tests to be useful downstream to optimize other message send sites. They also do not account for the better incremental compilation properties of exhaustive cone tests compared with exhaustive singleton class tests. Also, these metrics are based on probabilities, and so give a good indication of the relative costs of the two approaches. Absolute frequency information, either derived from profile data or estimated from the program's structure using static techniques [Wall 91, Wagner et al. 94], would also be useful, to determine if a particular message send site is deserving of substantial optimization effort.

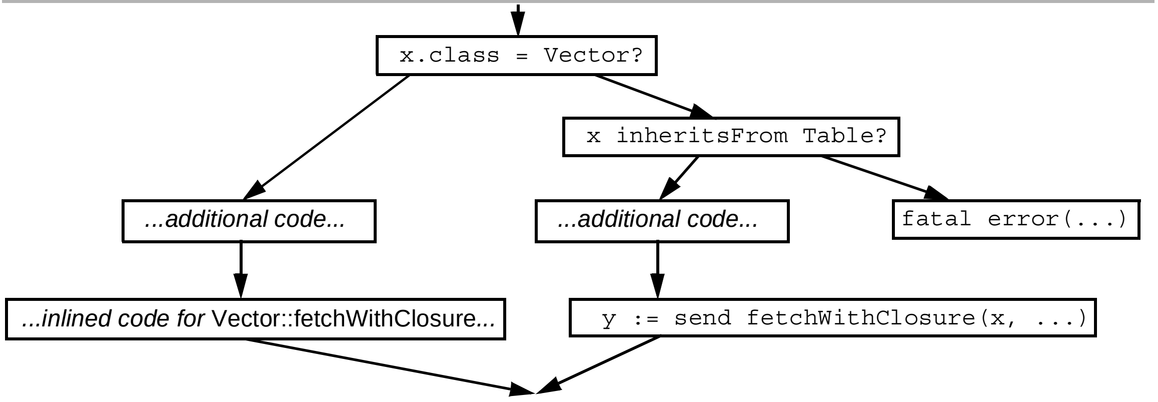
3.3.5 Check Strengthening

The situation illustrated in Figure 3.15 (c) (and reprised in Figure 3.15 (a)) arose because a call site downstream of the initial cone test desired more precise information than was garnered by the cone test. This necessitated the insertion of another, more precise test. This is undesirable if the case checked by the more precise test is more common than the other cases, since the common case path will then involve two tests: a less precise test, followed by a more precise test. By performing the more precise test first, we can reduce the number of tests on the common-case path from two to one. Ideally, in these cases the more precise test would simply be inserted eagerly, before the less precise test and the compilation process would continue. Unfortunately, detecting what sort of more precise information will be useful downstream is difficult, since it involves foreseeing how the downstream code is going to be optimized before actually attempting to optimize it. Instead, check strengthening can be applied to lazily create the desired effect, once it becomes clear that more precise information is necessary. When the compiler detects that it has introduced a more precise test downstream of a less precise test of the same value, as shown in Figure 3.15 (a), and that it is desirable to invert these tests, then the tests can be inverted by moving the precise test above the less precise test and duplicating the code between the precise test's old and new location. This transformation is illustrated in Figure 3.15 (b). This still leaves behind code to handle cases where x inherits from Table but is not a Vector. If x is almost always a Vector, then it may be preferable to end up with the code that results from just using profile-guided class prediction at the original call site, as illustrated in Figure 3.15 (d), but it is hard to revert the uncommon case of the control flow graph back to the unoptimized send once it has been optimized.

Strengthening of class tests is closely related to work by Kolte & Wolfe on eliminating array bounds checks in the context of Fortran programs [Kolte & Wolfe 95]. In their work, for bounds checks inside loops that involve linear functions of the loop induction variable, they developed an optimization that computed the strongest bounds check that will be required by any of the iterations in the loop, and hoisted a copy of this strong bounds check outside of the loop, so that it can be performed once. In their context, identifying downstream uses of the stronger information is easier, because the "downstream" uses that they examine are all executions of the same bounds checking code on different iterations of the loop. Since they restrict themselves to looking at bounds checks within the loop where the index function is a linear function of the induction variable, it is easier for them to eagerly compute the strongest check that will be needed downstream by the bounds checking code in the loop. In our domain,


x inheritsFrom Table?
...additional code...
fatal error(...)
x.class = Vector?
...inlined code for Vector::fetchWithClosure...
y := send fetchWithClosure(x, ...)

(a) Before check strengthening



(b) After check strengthening

Figure 3.15: Check strengthening

it is harder to compute the strongest information that will be needed downstream without performing something very close to the full optimization. Devising a scheme that optimizes message sends in the larger context of the rest of the control flow graph, to take into account downstream uses of information, and which does so without excessive compile time or code space costs, is a difficult problem but one worthy of future investigation.

3.4 Performance Evaluation

The impact of the optimization techniques described in this chapter depends to a large degree on the extent to which dynamic dispatching is used and is also influenced by aspects of the programming language in use. This section evaluates the performance impact of intraprocedural class analysis, class hierarchy analysis, and exhaustive class testing on a suite of medium-to-large object-oriented programs written in a variety of languages. The effects of the optimizations are examined both in isolation, and in conjunction with the other techniques and with profile-guided class prediction. The programs are written in Cecil, C++, Java, and Modula-3, four different languages with very different language characteristics. Section 3.4.1 discusses the languages and benchmarks in more detail. In an effort to understand what characteristics of a program are important in predicting the effect of the optimizations, Section 3.4.1 also develops a series of metrics that characterize the use of object-oriented features in these programs, and these metrics are used to explain and understand the performance effects of the various optimization techniques.

3.4.1 Benchmark Suite

A program's structure and the degree to which it uses object-oriented language features, such as inheritance and message sends, have a profound impact on the effectiveness of high-level optimizations such as class hierarchy analysis, exhaustive class testing, and profile-guided receiver class prediction. Therefore, before we present our experimental assessment of these techniques, we first define several metrics for describing object-oriented programs and use them to characterize our benchmark suite. These metrics attempt to quantify interesting properties of the program's internal structure as well as predict how much an application can be expected to benefit from a particular optimization. We considered a number of different metrics for characterizing our applications. After evaluating how well they captured the underlying program structure and usage of object-oriented language features, we selected the following metrics as being the most illuminating:

- *Number of Immediate Parents*: Measures the number of immediate parents of each class in the program; indicates the degree to which multiple inheritance is utilized.
- *Number of Immediate Children*: Measures the number of classes that directly inherit from each class in the program; indicates the branching factor (breadth) and "bushiness" of the class hierarchies.
- *Maximum Distance to Root of Inheritance Hierarchy*: The longest path from each class to the root of its inheritance hierarchy; indicates the depth of the class hierarchies used by the program. This metric and the number of immediate children metric are useful to get a sense of the kinds of class hierarchies used by each benchmark program.

- *Number of Applicable Methods*: Measures the number of applicable methods at a dynamically-dispatched call site, optionally weighted by the execution frequency of the call site. Class hierarchy analysis works well when many call sites have one applicable method, and exhaustive testing applies when a few methods are possible at a call site.
- *Class Test Efficiency*: Measures the fraction of calls at a call site that go to the most common receiver class at that call site, weighted by the execution frequency of the call site. This histogram will always be a subset of the one representing the dynamic number of applicable methods at a call site. Profile-guided class prediction can work well if the most common class at a call site is much more common than other classes; similarity between this histogram and the histogram for the dynamic number of applicable methods implies that high frequency call sites are dominated by a single receiver class.
- *Average Cycles Per Message Send*: Measures the average number of machine cycles elapsed between message sends, giving an indication of how frequently message sends are used in the program. As this value increases, we expect the overall performance impact of the optimizations to decrease.

Bieman and Zhao also used the first three of these metrics (and some additional ones) in their study of inheritance in C++ applications [Bieman & Zhao 95]. They utilized the metrics to assess the amount of code reuse through inheritance in large C++ programs; in contrast, we are interested in characterizing how the structure of the inheritance hierarchy affects the need for optimization. In previous work, we applied several metrics to measure the peakedness and stability of the profile data used to drive profile-guided receiver class prediction [Grove et al. 95].

We applied these metrics to a number of medium-to-large applications written in Cecil, Java, Modula-3, and C++. Table 3.4 summarizes several distinguishing language features. Several of these language characteristics have a large impact on the effectiveness of the object-oriented optimizations. Because of the much higher frequency of message sends in pure languages compared to hybrid languages, the impact of the object-oriented optimizations is much more dramatic. .Static type declarations are excellent fodder for class hierarchy analysis, so

Table 3.4: Language Characteristics

Language	Object Model	Typing	All Methods Virtual?	Multiple Inheritance?
Cecil	Pure	Dynamic[a]	Yes	Yes
Java	Mostly pure	Mostly static	Yes[b]	Yes[c]
Modula-3	Hybrid	Static	Yes	No
C++	Hybrid	Static	No	Yes

a. Cecil allows mixing statically- and dynamically-typed code, and running the static typechecker is optional. Thus, the optimizer ignores static type declarations and ensures type-safety through dynamic checks where needed.
b. `final` methods cannot be overridden, although they can override other methods.
c. Java supports multiple subtyping but only single code inheritance. Our number-of-parents metric indicates the total number of supertypes and superclasses of a class.

one would expect it to be quite effective in statically-typed languages. Table 3.5 describes the application suite and presents the results of applying the metrics to the programs. We use histograms to visually display the metrics; the height of each bar represents the percentage of all elements whose metric value corresponds to the bar's *x*-coordinate For C++, we also examined versions of two of the benchmarks where we hand-modified the programs to make all methods virtual. These programs are identified with an -av suffix, and represent an exploration of the costs of making all methods virtual in C++ under a variety of optimization techniques.

All of our benchmarks are substantial in size, with all at least 10,000 lines and most over 20,000 lines. The Cecil programs have the largest and deepest class libraries, with javac and m2tom3 also having deep class hierarchies, but all the programs except prover have at least a hundred classes and 400 methods. Cecil and Java programs used multiple inheritance or subtyping a moderate amount, while the C++ programs made little use of multiple inheritance.

Examination of the static number of call sites with only one applicable method would suggest that class hierarchy analysis could be effective in most of the benchmarks, but the version weighted by dynamic execution frequency indicates a somewhat lower expected benefit; eon and ktsim-av buck this trend. As expected, the C++ all-virtual programs have much greater number of call sites amenable to class hierarchy analysis than the regular versions of these benchmarks. About half the benchmarks have significant numbers of messages with two or three applicable methods, indicating that exhaustive class testing could be beneficial (although Vortex does not yet support this technique for C++ or Modula-3). Since the histograms reporting the efficiency of profile-guided class testing are usually similar to the dynamic number of applicable method histograms, profile-guided class prediction looks promising.

Table 3.5: Application Descriptions

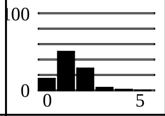
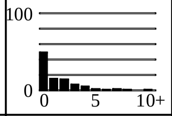
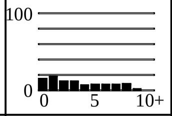
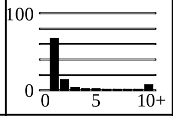
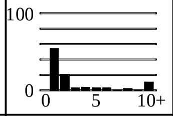
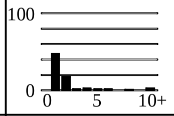
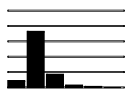
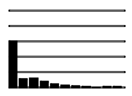
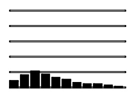
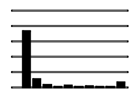
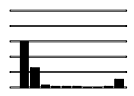
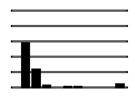
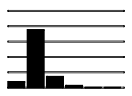
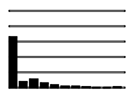
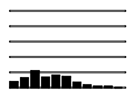
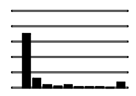
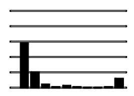
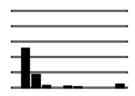
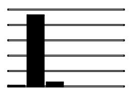
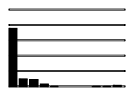
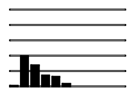
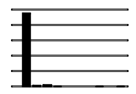
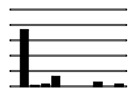
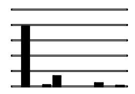
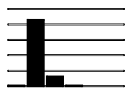
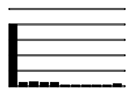
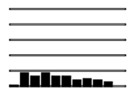
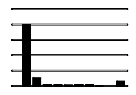
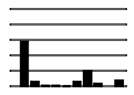
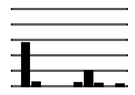
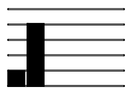
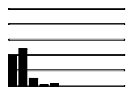
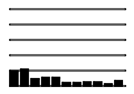
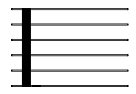
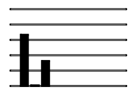
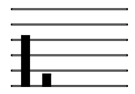
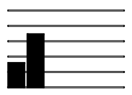
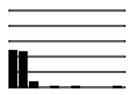
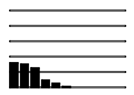
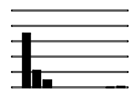
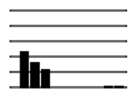
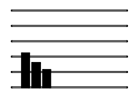
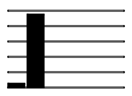
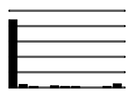
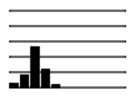
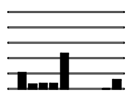
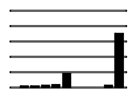
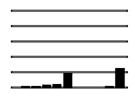
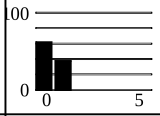
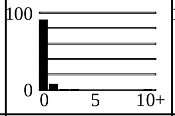
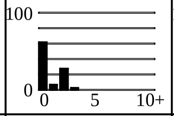
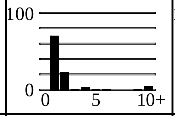
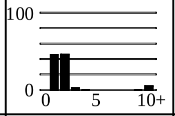
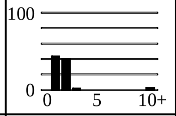
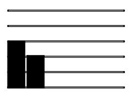
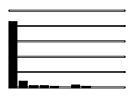
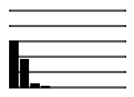
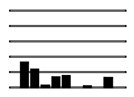
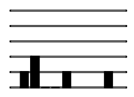
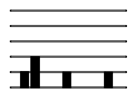
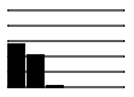
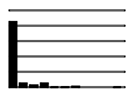
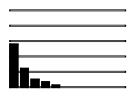
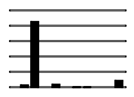
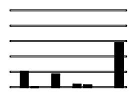
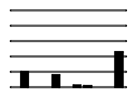
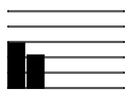
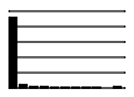
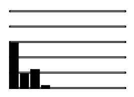
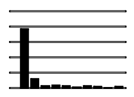
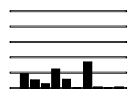
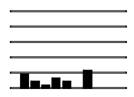
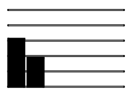
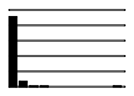
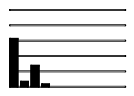
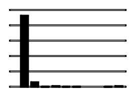
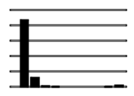
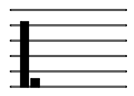
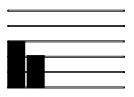
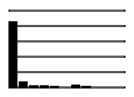
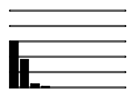
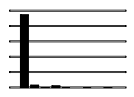
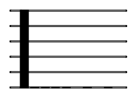
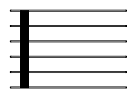
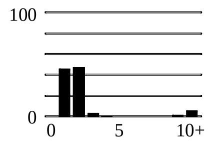
	Program	Description	Lines of Code	# of Classes	# of Dispatched Methods	% of classes with *x* immediate parents (x = 0..5)	% of classes with *x* immediate children (x = 0..9,10+)	% of classes with distance *x* from root of hierarchy (x = 0..9,10+)	% of message sends w/*x* applicable methods (x = 0..9,10+)		Class test efficiency for sends w/ *x* applicable methods (x = 0..9,10+)	Average cycles between starts of message sends
									Static	Dynamic		
Cecil	instr sched	Global instruction scheduler	2,400 +11,000 std. lib.	212	1640							127
	type-checker	Cecil typechecker	20,000 +11,000 std. lib.	609	4687							170
	vortex	Vortex optimizing compiler	68,500 +11,000 std. lib.	1306	10,062							197
Java	java_cup	Java parser generator	9,200 +12,200 library	124	484							165
	javac	Java compiler	25,400 +13,700 library	265	1465							240
Modula-3	m2tom3	Converts Modula-2 to Modula-3	18,005	105	432							479
	prover	Theorem prover	20,497	39	108							1522
	m3fe	Modula-3 front end	50,849	107	1170							2433

Table 3.5: Application Descriptions

	Program	Description	Lines of Code	# of Classes	# of Dispatched Methods	% of classes with *x* immediate parents (x = 0..5)	% of classes with *x* immediate children (x = 0..9,10+)	% of classes with distance *x* from root of hierarchy (x = 0..9,10+)	% of message sends w/*x* applicable methods (x = 0..9,10+)		Class test efficiency for sends w/ *x* applicable methods (x = 0..9,10+)	Average cycles between starts of message sends
									Static	Dynamic		
C++	ixx	Fresco IDL parser	11,600	119	791							575
	ktsim	memory system simulator	20,300	115	396							9001
	eon	Ray tracer	43,000	238	542							670
	porky	SUIF back-end optimizer	64,000	374	1257							220
C++ All Virtual	ixx-av	fresco IDL parser	11,600	119	977							146
	ktsim-av	memory system simulator	20,300	115	580							153

To the right is an enlarged copy of the histogram displaying the dynamic number of message sends with *x* applicable methods for the ixx benchmark. This histogram shows that roughly 42% of the messages sent by the program are sent at call sites where there is only 1 applicable method, 45% are sent from call sites with 2 applicable methods, and 5% are sent from call sites having 10 or more applicable methods.



The number of cycles between dispatches suggests that Cecil, Java, and the all-virtual C++ programs can expect the greatest impact from the optimizations, while Modula-3 programs can expect the least benefits from Vortex's message-level optimizations. With Cecil, optimization of other run-time overhead between dispatches, such as closure creations and the extra cost of multi-method dispatching, will increase the observed impact of Vortex's optimizations.

3.4.2 Results

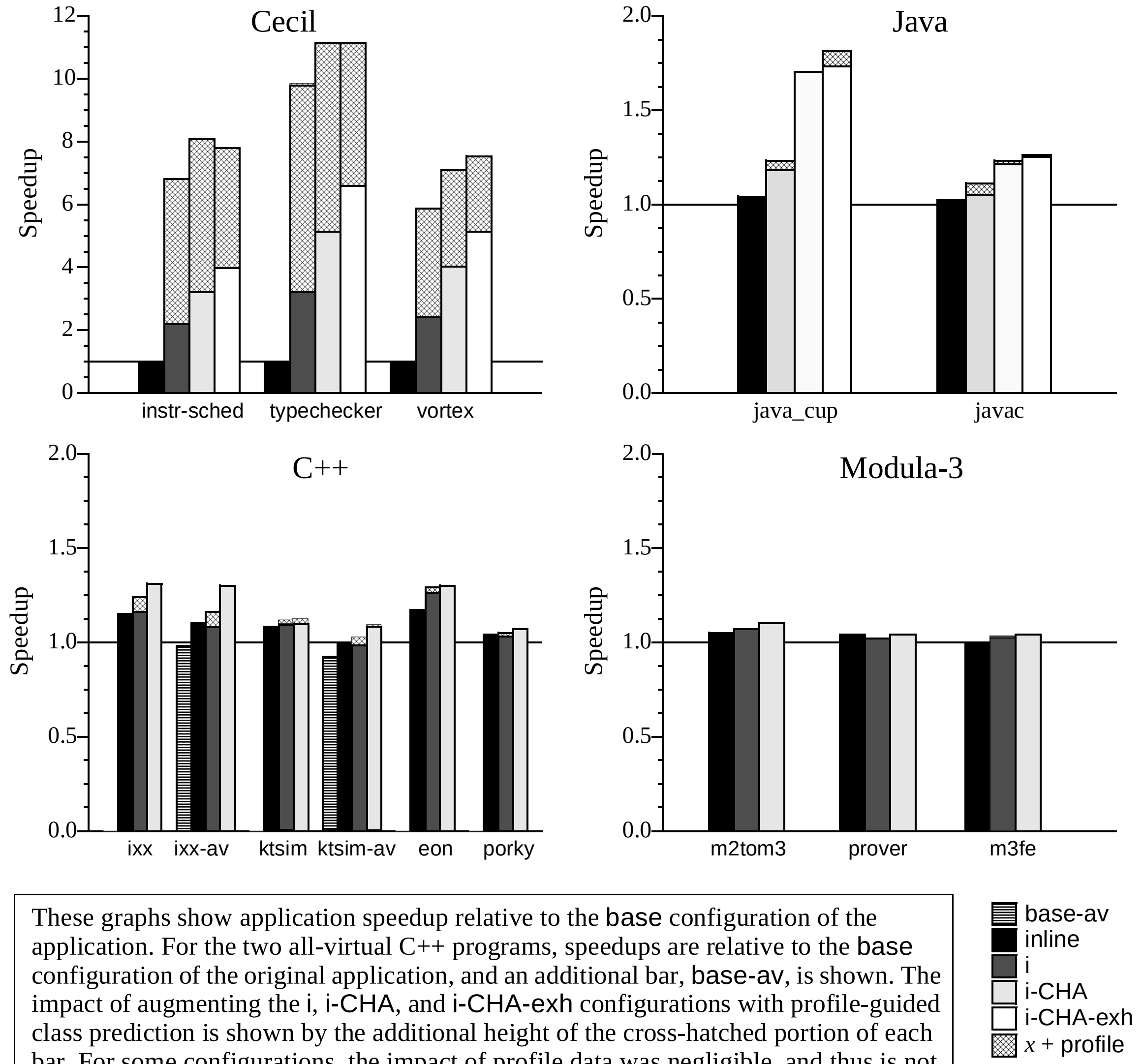
To evaluate the effectiveness of class hierarchy and exhaustive testing, and to compare these techniques against other common optimization techniques for object-oriented languages, we compiled the programs under several different configurations:

- **native**: Program is compiled by a native language compiler. For C++ programs we use g++ version 2.6.3 with options -O2 -finline-functions -msupersparc; for Modula-3 programs we use the DEC SRC Modula-3 compiler (which uses a modified version of gcc version 2.6.3 as its back-end) with the -O2 option. For Java the native configuration runs the java interpreter on the program's precompiled .class files. There is no separate native configuration for Cecil programs, since Vortex is currently the only Cecil compiler.
- **base**: Program is compiled by Vortex with a collection of traditional optimizations, including common subexpression elimination, dead assignment elimination, and dead store elimination, but without any inlining or optimization of dynamically-dispatched messages or class tests, and without any additional inlining of procedures beyond what the various language front-ends already performed. In all cases, the native language front-ends were invoked in modes that did as much optimization as possible. This meant that the C++ front-end performed standard intra-module inlining of function and non-virtual methods before translating the code to Vortex IL, that the Java front-end resolved and inlined method invocations of final methods, and that the Modula-3 front-end implemented cross-module procedure calls as direct procedure calls, rather than indirect procedure calls through a table. This was done to avoid overstating the effects of Vortex's optimizations. For C++ and Modula-3, the common subexpression elimination pass is not fully functional, so the base configuration in these languages uses only gcc optimizations with no Vortex-level optimizations.

- **inline**: base is augmented with cross-module inlining.
- **intra**: inline is augmented with intraprocedural class analysis, hard-wired class prediction for common messages (for Cecil programs only), splitting, and partial dead code elimination of closure creations.
- **intra+CHA**: intra is augmented with class hierarchy analysis.
- **intra+CHA+exh**: intra is augmented with class hierarchy analysis and exhaustive class testing. Exhaustive testing is only supported for Java and Cecil, so this configuration (and the intra+CHA+exh+profile configuration below) is only compared for these languages.
- **intra+profile**: intra is augmented with profile-guided receiver class prediction.
- **intra+CHA+profile**: intra is augmented with class hierarchy analysis and profile-guided receiver class prediction.
- **intra+CHA+exh+profile**: intra is augmented with class hierarchy analysis, exhaustive class testing, and profile-guided receiver class prediction.

In all Vortex configurations, C code was generated and then compiled by gcc version 2.6.3 with options -O2 -msupersparc. By producing C code, and compiling it with the same compiler used in the native configuration for C++ and Modula-3[7], we attempt to minimize distortions produced by aspects of Vortex compilation that are orthogonal to the high-level optimizations we are interested in studying. For the C++ and Modula-3 programs, with the exception of prover (13%) and porky (39%), all base times were within 8% of their native time. The large performance difference between the native and base configurations for porky is explained by porky's frequent assignments of small 4 and 8 byte structures. Currently, Vortex does not attempt to optimize structure assignments, compiling them down to calls to `bcopy`. The gcc compiler compiles down small structure assignments to a sequence of inline loads and stores. For Modula-3, our base configuration is losing some performance relative to native due to compilation to C; in particular, nested procedures are not implemented as efficiently in base as in native. For Java, the native (interpreted) version of javac was 6 times slower than Vortex's

7. The native Modula-3 compiler actually uses the back end from the gcc compiler version 2.7.2. However, the differences in code quality between versions 2.6.3 and 2.7.2 of the compiler are relatively minor.



These graphs show application speedup relative to the base configuration of the application. For the two all-virtual C++ programs, speedups are relative to the base configuration of the original application, and an additional bar, base-av, is shown. The impact of augmenting the i, i-CHA, and i-CHA-exh configurations with profile-guided class prediction is shown by the additional height of the cross-hatched portion of each bar. For some configurations, the impact of profile data was negligible, and thus is not visible above.

Figure 3.16: Performance impact of Vortex optimizations

base configuration, demonstrating the performance advantage of compilation over interpretation.

All run-time measurements are CPU times (user time + system time) gathered on a lightly-loaded SPARCStation-20/61 with 128MB RAM, taking the median of 11 runs. Variations in CPU time from run to run of 10% are normal for this machine. Figure 3.16 presents selected application speedup data; the complete set of application execution times and the dynamic number of message sends during each program execution can be found in Appendix A.

As expected, Vortex's optimizations had the largest impact on the Cecil programs, improving their performance by an order of magnitude over the base configuration. The relatively larger improvement of Cecil programs can be partly attributed to the higher cost of method dispatches, which support a more flexible dispatching model based on multi-methods and predicate classes, and on the greater indirect benefits from inlining due to eliminating closure creations. The higher direct and indirect cost of a dispatch makes optimizations to eliminate them more effective than in other languages with relatively cheaper dispatches. Profile-guided class prediction was the single most important optimization for Cecil, but each technique resulted in a non-trivial improvement. Although less spectacular than the Cecil results, many of the C++ and Java programs obtained speedups on the order of 25-35%; m2tom3 showed the largest speedup of the Modula-3 programs, gaining about 5% from cross-module inlining and an additional 4% from the object-oriented optimizations. In the non-Cecil programs, which optimizations were most effective varied from one application to the next.

However, the performance gains obtained by applying these optimizations does not come without cost; both compile time and compiled code space typically increase as a result. In the worst case, observed for some of the C++ programs, compilation time doubled between base and inline, and doubled again between inline and any of the other more optimized configurations. However, Vortex is a research compiler, designed for ease of extension rather than compilation efficiency, and so the magnitude of the compile time increases may not be indicative of what would be seen if these techniques were implemented in a production compiler. In contrast, code space costs were modest. For the C++ programs, compiled code space grew by 3% to 20% over the base configuration. For the other three languages, code space changes ranged from –10% to +4% over base.

The failure of prover and m3fe to benefit more substantially from these optimizations was foreshadowed by the average cycles between the starts of message sends metric (1522 and 2433 respectively): those two Modula-3 programs spend little of their time performing dynamic dispatching, and the costs of dispatching are low. Overall, the dynamic number of applicable methods metric was a good predictor of the effectiveness of class hierarchy analysis. The three programs which saw the smallest reduction in message sends (m3fe, porky, and eon), also had the smallest dynamic percentage of call sites with a single applicable method. However, this metric was not a perfect predictor; ktsim looked very similar to eon, but a large fraction of its message sends were eliminated by class hierarchy analysis. The class test efficiency metric was suggestive of the effectiveness of receiver class prediction, but was not always accurate because Vortex only inserts class tests based on profile information when the target method is small enough to be inlined, and the metric does not account for target method size.

3.4.3 Potential Improvements from More Powerful Static Analyses

When all the message send optimizations examined in Section 3.4.2 have been applied to these programs, the Cecil programs still exhibit substantially higher message send frequencies than the programs written in the other languages (around 150,000 messages/second, versus 80,000/second for Java, 50,000 for the C++ programs, and 30,000/second for Modula-3). Class hierarchy analysis and exhaustive testing, coupled with profile-guided class prediction, are successful at optimizing more than 90% of the message sends in Cecil programs, but it is possible that more powerful static analyses or information provided by a static type system could optimize away substantial portions of the remaining dispatches, as well as eliminate existing class and cone tests inserted to optimize other dispatches. Of particular interest is the impact of dynamic typing on the Cecil results, since many message sends that remain could be optimized if the compiler had static type information to aid in the class hierarchy analysis process. In the absence of static type information, the compiler could also use a more powerful interprocedural class analysis algorithm, such as Agesen's Cartesian Product Algorithm [Agesen 95] or Plevyak & Chien's iterative interprocedural algorithm [Plevyak & Chien 94], to obtain more precise information about the classes appearing at each message send.

To explore how much room for improvement there is from either static typing or interprocedural class analysis algorithms, we utilized profile information as an oracle, to give the compiler "perfect" information about what classes appeared at each call site in the program. The perfect class information was collected on a per call-site basis, effectively providing an upper bound on the effectiveness of any interprocedural class analysis algorithm that utilizes a single analysis context per-call-site in the program (effectively a 0-CFA-style algorithm, in the terminology of Shivers [Shivers 88]). This experiment also gives a crude upper bound on how much benefit the compiler might obtain if Cecil possessed a non-parameterized static type system, such as that found in Java.[8]

To perform this experiment, the perfect profile was collected, and the programs were recompiled using this perfect profile information as an oracle that was consulted whenever static class information was needed for optimizing a message send. Portions of the program that the profile information indicated would not be executed were eliminated from the executable, including both unreachable routines as well as unreachable paths within live routines. Of course, because this static information is derived from profiling the program to determine what classes actually appear at each call site, the resulting executable is only guaranteed to work for exactly the same input as was used in the original profile run. As such, it provides an upper bound on the effectiveness of any context-insensitive class analysis. The results of this experiment are shown in Table 3.6.[9]

Table 3.6: Impact of perfect static information on Cecil programs

Program	intra+CHA+exh+profile			intra+CHA+exh+profile with perfect static info		
	Execution time (secs)	Dispatches	Class tests Cone tests	Execution time (secs)	Dispatches	Class tests Cone tests
scheduler	2.03	393,150	1,824,363 (class) 427,822 (cone)	1.43 (0.70)	126,968 (0.32)	322,341 (0.18) 7,050 (0.02)
typechecker	31.39	6,377,108	19,078,487 2,558,575	21.54 (0.69)	4,966,904 (0.78)	7,923,631 (0.41) 426,601 (0.16)
compiler	460	57,927,979	175,856,901 23,655,003	328 (0.71)	40,046,750 (0.69)	61,609,110 (0.35) 5,028,615 (0.21)

Given perfect static class information of this form, the three Cecil programs sped up by an average of 30%. The number of dynamic dispatches was reduced by roughly 25% for the two large programs, but the programs exhibited a 2.5- to 5-fold reduction in the number of class tests and a 5- to 60-fold reduction in the number of cone tests executed. This 30% improvement in execution time suggests that using the results of a context-insensitive interproce-

8. Cecil's actual type system is a substantially more powerful, parameterized system based on F-bounded polymorphism. However, the static type system thrust of the Cecil research project is ongoing, and a working typechecker does not yet exist for the whole of the type system that has been designed. Although type declarations have been put into all three of the Cecil programs from the beginning of their development, the typechecker is not functioning to verify that these declarations are correct, and therefore the compiler is forced to ignore them, despite their potential as a rich source of static information.

9. The programs were compiled with a slightly different version of the compiler and runtime system than was used to gather the results presented in Section 3.4.2, so the results for the intra+CHA+exh+profile configuration are not identical to those presented in Section 3.4.2.

dural class analysis algorithm to optimize message sends will improve the performance of Cecil programs only moderately. More powerful, context-sensitive algorithms or procedure specialization based on static analyses might provide more substantial performance improvements. This does not mean, however, that interprocedural optimizations will be of only limited benefit. Performing other optimizations based on the results of an interprocedural class analysis, such as code duplication for parameterized data structures, or optimizing data structure representations, might result in substantial performance benefits: the results presented here only examine the ability of a perfect context-insensitive algorithm to provide further performance benefits by optimizing message sends.

3.5 Other Issues

Supporting incremental programming changes in the face of whole-program optimization, and dealing with situations where parts of the program are not available to the compiler, are two important areas for making these techniques practical. These issues are discussed in Section 3.5.1 and Section 3.5.2, respectively.

3.5.1 Incremental Programming Changes

Class hierarchy analysis might seem to be in conflict with incremental compilation: the compiler generates code containing embedded assumptions about the structure of the program's class inheritance hierarchy and method definitions, and these assumptions might change whenever the class hierarchy is altered or a method is added or removed. A simple approach to overcoming this obstacle is to perform class hierarchy analysis and its dependent optimizations only after program development ceases. A final, batch optimizing compilation could be applied to frequently-executed software just prior to shipping it to users, as a final performance boost. This assumes that the points at which the performance boost can be clearly identified (i.e. shipping a product), and that they are infrequent enough so that the compile-time costs of recompiling the entire program are not an issue.

Class hierarchy analysis can be applied even during active program development, however, if the compiler maintains enough intermodule dependency information to be able to selectively recompile those parts of a program invalidated after some change to the class hierarchy or the set of methods. This is the approach followed in the Vortex compiler, and Chapter 5 contains a detailed discussion of how the compiler keeps track of recompilation dependencies and how these are used to determine what code needs to be recompiled as a result of programming changes.

3.5.2 Optimization of Incomplete Programs

Class hierarchy analysis can only be applied when the compiler either has access to the entire class hierarchy of the program, or when it at least knows which pieces of the class hierarchy it does not have information about. Obviously, it is most effective in situations where the compiler has access to the source code of the entire program, since the whole inheritance hierarchy can be examined and the locations of all method definitions can be determined. As a side benefit, having access to all source code also provides the compiler with the option of inlining any routine once a message send to the routine has been statically-bound. Although today's integrated programming environments make it increasingly likely that the whole program is available for analysis, there are still situations in which having source code for the entire program is unrealizable. In particular, a library may be developed separately from client applications, and the library developer may not wish to share source code for the library with clients. For example, many commercial C++ class libraries provide only header files and compiled `.o` files and do not provide complete source code for the library.

Fortunately, having full source code access is not a requirement for class hierarchy analysis. As long as the compiler has knowledge of the class hierarchy and where methods are defined in the hierarchy (but not their implementations), class hierarchy analysis can still be applied, and this information usually is available in the header files provided for the library. When compiling the client application, the compiler can perform class hierarchy analysis of the whole application, including the library, and statically bind calls within the client application. If source code for some methods in the library is unavailable, then statically-bound calls to those methods simply will not be able to be inlined. Static binding alone can provide significant performance improvements, particularly on modern RISC processors without branch target buffer hardware, where dynamically-dispatched message send implementations stall the hardware pipeline [Calder & Grunwald 94]. A recent study indicates that a suite of moderate-sized C++ programs spend 5.2% of their time in the virtual function dispatching sequence alone, and that this number increased to more than 13% if all methods are virtual [Driesen & Hölzle 96]. Furthermore, once a send has been statically-bound some optimizing linkers are able to optimize static calls by inlining the target routine's machine code at link time [Srivastava & Eustace 94, Fernandez 95, Larus & Schnarr 95], although the resulting code is not as optimized as what could be generated had the compiler done the inlining.

Using class hierarchy analysis when compiling a library in isolation is more difficult, since the client program might create subclasses of library classes that override methods defined in the library. The sealing approach of Dylan and Trellis can provide the compiler with information about what library classes won't be subclassed by client applications, supporting class hierarchy-based optimizations for those classes at the cost of reduced extensibility. Alternatively, the compiler could choose to compile specialized versions of methods applicable only to classes present in the library. For example, in a data structure library, the compiler could compile specialized versions of methods for array, string, hash table, and other frequently-used classes; generic versions of methods would also be compiled to support any subclasses of these library classes defined by client applications. Chapter 4 describes a profile-guided algorithm that examines the potential targets of sends in a routine and derives a set of profitable specializations based on where in the class hierarchy these target routines are defined. This specialization algorithm detects call sites where class hierarchy analysis is insufficient to statically-bind message sends, and produces versions of methods specialized to truncated cones of the class hierarchy. Empirical measurements indicate that this algorithm applied to a complete 52,000-line Cecil program improves performance by up to 10% or more with less than 5% compiled code space increase, although we would expect a larger relative space overhead if the algorithm were applied to a library in isolation.

3.6 Related Work

Recent work by Fernandez has investigated using link-time optimization of Modula-3 programs to convert dynamic dispatches to statically bound calls when no overriding methods were defined [Fernandez 95]. This optimization is similar to class hierarchy analysis. A typical problem with link-time optimizations is that, because they usually operate on machine code, the representation of the program is too low-level to perform high-level optimizations such as class hierarchy analysis. To overcome this, Fernandez's system is based on a compiler that outputs a higher-level intermediate form, rather than native machine code. The linker then combines together the intermediate code for the various object modules, and performs optimizations and then code generation for the whole program. However, because all optimizations and code generation are now done at link-time, care must be taken to prevent linking from becoming a bottleneck in the edit-compile-debug cycle. In effect, Fernandez's "linker" look very much like a whole-program-optimizing compiler that compiles the whole program every time it is linked; Fernandez acknowledges that this design penalizes turnaround time for small programming changes. Additional optimizations would only increase this penalty. In contrast, Vortex's approach combines whole-program optimization at compile-time with a selective invalidation mechanism to support incremental recompilation, fast linking, and compile-time optimization of call sites where source code of target methods is available. However, given the appropriate selective invalidation machinery, Fernandez's approach and the approach used in Vortex are very similar: both are trying to perform optimization when the whole program is available.

Diwan, Moss, and McKinley have built an optimizing Modula-3 system based on whole-program analysis [Diwan et al. 96]. Their system includes several techniques similar to the Vortex compiler's techniques, including intraprocedural static class analysis and class hierarchy analysis. They include neither profile-guided receiver class prediction nor selective specialization, and their intraprocedural analysis does not narrow class sets on paths downstream of run-time class tests. They do include a form of interprocedural class analysis, where the intraprocedural solution is used to construct a call graph on which the interprocedural analysis is based. Unfortunately, while their analyses resolve many of the static call sites in their benchmarks (ignoring the possibility of NULL error invocations), the bottom-line speed-up of their benchmarks was less than 2%, due in large part to the infrequency of dynamically-dispatched calls in their benchmarks.

Aigner and Hölzle implemented a prototype system to compare class hierarchy analysis and profile-guided receiver-class prediction for C++ programs [Aigner & Hölzle 96]. Their compiler first combines all C++ input files into a single monolithic file, then applies transformations to the input file, and produces an output C++ file which is then compiled by a regular C++ compiler. They substantially reduce the number of dynamic dispatches in their C++ benchmarks, and achieve modest bottom-line speed-ups as a result. Their system does not attempt to support separate or incremental recompilation, and by relying on the underlying C++ compiler to perform inlining, their analyses can suffer by not having accurate knowledge of the code after inlining.

Srivastava has developed an algorithm to prune unreachable procedures from C++ programs at link-time [Srivastava 92]. Although the described algorithm is only used to prune code and not to optimize dynamic dispatches, it would be relatively simple to convert some virtual function calls into direct procedure calls using the basic infrastructure used to perform the procedure pruning.

Interprocedural class analysis algorithms are an important area of current research [Palsberg & Schwartzbach 91, Oxhøj et al. 92, Plevyak & Chien 94, Pande & Ryder 94, Agesen 95, Grove 95]. By examining the whole program and solving an interprocedural data flow problem to determine the set of classes that might be stored in each program variable, these algorithms can provide more accurate class sets than intraprocedural static class analysis or class hierarchy analysis. However, current interprocedural class analysis algorithms are relatively expensive to run, assume access to the source code of the entire program, including method bodies, and are not incremental in the face of programming changes. Nevertheless, as these algorithms mature, it will be interesting to compare the run-time benefits and compile-time costs of interprocedural class analysis against class hierarchy analysis and the other techniques examined in this thesis. Agesen and Hölzle have done an initial investigation into this area by comparing the effectiveness of profile-guided receiver class prediction with the Cartesian Product Algorithm, an interprocedural class analysis, for a suite of small-to-medium-sized Self programs. They found that both were effective techniques but that both techniques had benchmarks on which they fared much better than the other technique [Agesen & Hölzle 95]. In later work, they reported that combining the two techniques did not seem to have much of a synergistic effect: the performance improvement of combining the techniques was usually no better than the better of the two techniques applied individually [Agesen & Hölzle 96].

3.7 Summary

Class hierarchy analysis is a promising technique for eliminating dynamically-dispatched message sends automatically. Unlike language-level mechanisms such as non-virtual functions in C++ and final methods and classes in Java, class hierarchy analysis improves performance while preserving the source-level semantics of message passing and the ability for clients to subclass any class. To integrate class hierarchy analysis effectively into existing static class analysis frameworks, we introduced the cone representation for a class and its subclasses and constructed applies-to sets for each method to support compile-time method lookup in the presence of cones and other unions of classes. Cones also provide a means for static type declarations to be integrated into static class analysis. Exhaustive class testing provides additional benefits, by permitting the inlining of message sends with limited degrees of polymorphism through the use of relatively fast inline tests. Class hierarchy analysis has been implemented in the Vortex compiler since the Spring of 1994, and exhaustive class testing has been implemented since the Spring of 1996. In Vortex, class hierarchy analysis is always performed as a matter of course, and intermodule dependency links support selective recompilation. Vortex is itself an 80,000-line Cecil program, undergoing rapid continuous development and extension, providing some evidence that class hierarchy analysis is compatible with a program development environment.

Chapter 4

Selective Specialization

The previous chapter described class hierarchy analysis, a technique that converts dynamically-dispatched messages into statically-bound calls, when it can be determined that only a single method can be invoked by a message send. It also described exhaustive class testing, which handles cases where only a few reasonably-small methods can be invoked by a message send. However, some message sends, notably those where a large number of methods could be invoked, defy optimization by either of these techniques. One technique for optimizing such message sends, by improving the precision of class information, is to compile multiple specialized versions of a method, each applicable to only a subset of the possible argument classes of the method. By choosing the specialized subsets judiciously, the compiler gains more precise information about the potential classes of the receiver and other arguments within each of the specialized versions of the method, and many message sends that are sent to the receiver or to the arguments of a method can be statically bound that could not be in the unspecialized version.

Some compilers for object-oriented languages implement a restricted form of specialization called *customization*, in which a specialized version of a method is compiled for each possible receiver class, and methods are never specialized for arguments other than the receiver [Chambers et al. 89]. Although this strategy yields substantial performance improvements, it can also lead to substantial increases in code size, especially for large programs with deep inheritance hierarchies and a large number of methods.

This chapter presents a goal-directed algorithm to identify profitable specializations that combines dynamic profile data with static information about the program's class hierarchy to limit the application of specialization to those cases where it is likely to provide substantial performance improvements. Unlike customization, the algorithm also is suitable for use in object-oriented languages with multi-methods. Our results show that the algorithm simultaneously increases program performance and decreases the compiled code space requirements over several alternative compilation and specialization strategies, including customization.

The next section provides an example motivating why specialization is an important optimization technique and discusses why existing specialization techniques are inadequate. Section 4.2 describes our algorithm for selective specialization. Its effectiveness is evaluated in Section 4.3, and Section 4.4 discusses related work.

4.1 A Motivating Example

To illustrate the issues involved in specialization, Figure 4.1 presents a somewhat expanded example of the `Set` class hierarchy that was first presented in Section 2.1. As before, the example has different subclasses of `Set` to implement different set representations.[1] The abstract base class provides operations to perform generic set operations such as union, intersection, and overlaps testing, and relies on dynamically-dispatched `do` and `includes` messages, with versions of these operations implemented in each subclass.

Although writing the code in this fashion, where many operations are factored into the abstract `Set` class, offers software engineering advantages, it leads to inefficient code if compiled naively. Compiling a single version of the `overlaps` method that is general enough to apply to all possible `Set` representations results in code that has substantial dynamic dispatching overhead. Even with a technique such as class hierarchy analysis, the `self.`***`do`***`(λ(elem){...})` message must be dynamically dispatched, since which implementation of `do` will be invoked cannot be determined statically and in fact can vary at run-time. Furthermore, within the closure, the send of `set2.`***`includes`***`(elem)` must also be dynamically dispatched, since there is insufficient static information about the class of `set2` to enable static binding to a single `includes` method. The insertion of predictive class tests might be able to optimize these message sends in some cases, but if the number of target methods or classes appearing at the call site is substantial or if no one class clearly is more frequent than other classes, then inserting predictive tests may not actually improve performance (and in fact can slow the program down). So, in many cases, the compiler is forced to

1. In this syntax, `λ(arg1:type){` *... code ...* `}` is a closure that accepts one argument, and `λ(type):type` is a static type declaration for such a closure. Formal parameters of the form `name@class` mean that the method dispatches on the specified class in that argument position. Dynamically-dispatched message sends are shown in ***`this`*** font.

```
class Set;
method overlaps(set1@Set, set2@Set):bool {
  set1.do(λ(elem){ if set2.includes(elem) then return true; });
  return false;
}
method includes(self@Set, elem):bool {
  ... a default includes method: subclasses can override to provide a more efficient implementation ...
  self.do(λ(elem2){ if elem.equal(elem2) then return true; });
  return false;
}
.. other set operations such as intersection, union, etc. ...

class ListSet isa Set;
method do(self@ListSet, body:λ(elem):void):void {
  ... code to iterate over elements of t, evaluating closure body on each element ...
}

class HashSet isa Set;
method do(self@HashSet, body:λ(elem):void):void { ... }
method includes(self@HashSet, elem):bool {
  ... a more efficient implementation of includes: hash the element and see if it is in that bucket ...
}

class BitSet subclasses Set;
method do(self@BitSet, body:λ(elem):void):void { ... }
method includes(self@BitSet, elem):bool {
  ... a more efficient implementation of includes: test the appropriate bit position ...
}
... more efficient versions of overlaps, union, intersection, etc. when operating on two BitSets ...
method overlaps(set1@BitSet, set2@BitSet):bool { ... }
```

Figure 4.1: A set abstraction hierarchy

leave such dynamic dispatches unoptimized, and as usual, this has both direct and indirect costs. The direct cost of performing the dynamic dispatching is substantial, but having the messages be dynamically dispatched also prevents other optimizations, such as inlining, which often has an even greater effect on code quality. For example, because the ***do*** message send was not able to be statically-bound and inlined, the closure argument to ***do*** must be created at run-time and invoked as a separate procedure for each iteration.

One way of improving the optimization of some of these dispatches is to compile multiple, specialized versions of source routines, each applicable to different subsets of the possible receiver and/or argument classes to the routine. *Customization* is a simple specialization scheme used in the implementations of some object-oriented languages, including Self [Chambers & Ungar 89, Hölzle & Ungar 94], Sather [Lim & Stolcke 91], and Trellis [Kilian 88]: a specialized version of each method is compiled for each class inheriting the method. Within the customized version of a method, the exact class of the receiver is known, enabling the compiler to statically-bind messages sent to the receiver formal parameter (`self`). In our example, a specialized version of `overlaps` would be compiled for `ListSet` and `HashSet`, and this would allow the `self.`***do***`(λ(elem){...})` to be statically-bound to the appropriate implementation in each of these versions (`BitSet` provides an overriding version of `overlaps`, and therefore uses this method definition for customization purposes). For example, in the `ListSet` version of `overlaps`, the `self.`***do***`(λ(elem){...})` message can get inlined and optimized down to a simple `while` loop. Other operations defined on `Set`, such as `union` and `intersection`, would be similarly customized. Because sends to `self` tend to be fairly common in object-oriented programs, customization is effective at increasing execution performance: Self programs run 1.5 to 5 times faster as a result of customization, and customization was one of the single most-important optimizations included in the Self compiler [Chambers 92]. Lea hand-simulated customization in C++ for a Matrix class hierarchy, showing an order-of-magnitude speed-up, and argued for the inclusion of customization in C++ implementations [Lea 90].

Unfortunately, this simple strategy for specialization suffers from the twin problems of *overspecialization* and *underspecialization*, because specialization is done without considering whether or not the more precise information provided by a customized routine is actually useful for optimization. In many cases, multiple specialized versions of a method are virtually identical and could be coalesced without a significant impact on program performance. For large programs with deep inheritance hierarchies and many methods, producing a specialized version of every method for every potential receiver class leads to serious code explosion. In the presence of large, reusable libraries, we expect applications to use only a subset of the available classes and operations, and some of those only infrequently, and consequently simple customization is likely to be impractical.

In systems employing dynamic compilation, such as the Self system [Chambers & Ungar 89], customization can be done lazily by delaying the creation of a specialized version of a method until the particular specialized instance is actually needed at runtime, if at all. This strategy avoids generating code for *class* × *method* combinations that are never used, but such systems can still have problems with overspecialization if a method is invoked with a large number of distinct receiver classes during a program's execution or if a method is invoked only rarely for particular receiver classes. Moreover, dynamic compilation may not be a suitable framework for all programming systems because of the need for an optimizing compiler and some representation of the program source to be included in the program's runtime system.

In addition to overspecialization, the simple customization approach suffers from *underspecialization*, because methods are never specialized on arguments other than the receiver. In many cases, considerable benefits can arise from specializing a method for particular argument classes. In our example, specializing the `overlaps` method for a particular class of the `set2` argument could have considerable performance benefits, since it would allow the `set2.`***`includes`***`(elem)` message inside the loop to be statically-bound and potentially inlined. By specializing on an argument other than the receiver, clients of `overlaps` will have to be compiled to choose the appropriate specialized version of `overlaps`, perhaps by inserting additional class tests or table lookups at run-time, but such tests occur only once per `overlaps` operation. Compared to incurring a dynamic dispatch for every element of the set, this overhead to select the appropriate specialization can be minor. In effect, specialization can hoist dynamic dispatches from frequently executed sections of code to less frequently executed sections, often across multiple procedure boundaries.

4.2 Selective Specialization

To address the problems of overspecialization and underspecialization, our work focuses on techniques for specializing only where the benefits are large. Rather than specializing exhaustively, our algorithm exploits dynamic profile data to specialize heavily-used methods for their most beneficial argument classes, preserving a single, general-purpose version of the method that works for the remaining cases. Moreover, our algorithm identifies when a specialization can be shared by a group of classes without great loss in performance, further reducing code space costs.

Our algorithm is based on a general framework in which a method can be specialized for a tuple of class sets, one class set per formal argument, including the receiver. For example, in the `Set` example we would describe one potential specialization of the `overlaps` method using the tuple <{`ListSet,HashSet`},{`HashSet`}>. An argument class set can include multiple classes when a specialization is shared by several classes. This framework has two advantages over previous schemes:

- Methods can be specialized to obtain more precise information about any formal argument, not just the receiver argument as with customization.
- Multiple classes can share a single specialized method, rather than generating a specialized copy for each individual class.

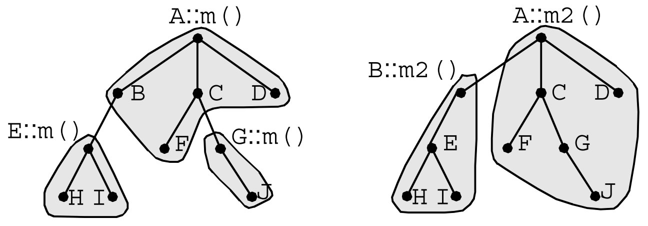
However, this general model places few constraints on specialization, requiring guidance in order to determine which potential specializations are profitable. Our algorithm relies on two sources of information to guide the specialization process:

- We exploit information about the program's class hierarchy to identify groups of classes that still allow static binding of messages.
- We use profile information about how many times each dynamically-dispatched call site invoked each different method to select the most important specializations in program hot spots. This form of profile information is a subset of that gathered by the `gprof` program [Graham et al. 82].

The algorithm will be illustrated with the class hierarchy and method definitions shown in Figure 4.2 and the weighted call graph shown in Figure 4.3. The class hierarchy consists of ten classes, with method `m` implemented only in classes `A`, `E`, and `G`, method `m2` implemented only in classes `A` and `B`, and methods `m3` and `m4` implemented only in class `A`. The shaded regions show the "equivalence classes" that all invoke the same implementation of `m` and `m2` (note that the illustrations in Figure 4.2 are two views of the same underlying inheritance hierarchy).

The algorithm is shown in Figure 4.4. Set operations on tuples are defined to operate point-wise on the tuple elements. The algorithm relies on the presence of a weighted program call graph constructed from the profile data that describes, for each call site in the program, the set of methods invoked and the number of times each was invoked (a call site can have multiple arcs due to dynamic dispatching). Given an arc in the call graph, *Caller*(*arc*) gives the calling method, *Callee*(*arc*) gives the called method, *CallSite*(*arc*) identifies the message send site within the caller, and *Weight*(*arc*) gives the execution count of the arc. For the arc labelled α in Figure 4.3, *Caller*(α) is `A::m4`, *Callee*(α) is `B::m2`, *CallSite*(α) is the send `arg2.`***`m2`***`()` within `m4`, and *Weight*(α) is 550.

The algorithm exploits information about the class hierarchy through the use of the *AppliesTo* information computed as part of class hierarchy analysis (see Section 3.2.2 for more details). *AppliesTo*(*meth*)returns the tuple of the set of classes for each formal argument for



```
method m3(self@A, arg2@A) { self.m4(arg2); }
method m4(self@A, arg2@A) { self.m(); arg2.m2(); }
                               1         2
```

Figure 4.2: Example class hierarchy

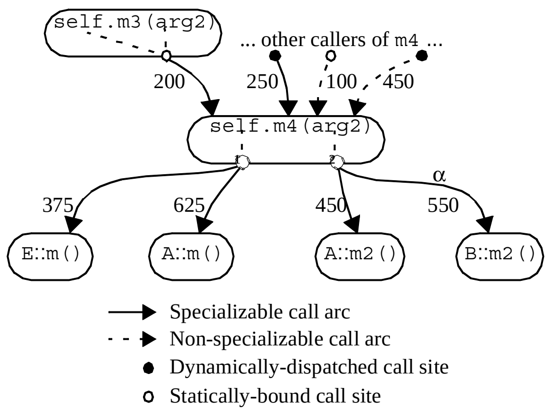

self.m3(arg2)
... other callers of m4 ...
200
250
100
450
self.m4(arg2)
375
625
450
α
550
E::m()
A::m()
A::m2()
B::m2()
Specializable call arc
Non-specializable call arc
Dynamically-dispatched call site
Statically-bound call site

Figure 4.3: Example call graph (with dynamically-weighted arcs)

which the method *meth* could be invoked (excluding classes that bind to overriding methods). The shaded "equivalence classes" regions in the example above identify the *AppliesTo* information for each of the m and m2 methods. For example, *AppliesTo*(method E::m)= <{E,H,I}>. For singly-dispatched languages, computing *AppliesTo* for each method is fairly straightforward. For multiply-dispatched languages, there are some subtleties in performing this computation efficiently; details of how this can be done can be found in Section 3.2.3.

The goal of the algorithm is to eliminate dynamic dispatches for calls from some method by specializing that method for particular classes of its formal parameters. By providing more precise information about the classes of a method's formals, the algorithm attempts to make more static information available to dynamically-dispatched call sites within the method to enable the call sites to be statically bound in the specialized version. The simplest case in which specializing a method's formal can provide better information about a call site's actual

Profile info:
Caller(*a*), *Callee*(*a*), *CallSite*(*a*), *Weight*(*a*) - give caller, callee, call site, and frequency for call graph arc *a*

Source info:
AppliesTo(`method m(`f_1`,..,`f_n`)`) = *n*-tuple of sets of classes for $f_1, \ldots, f_n$ for which `m` might be invoked
PassThroughArgs⟦`msg(`a_1`, ..., `a_n`)` *in* `method m(`f_1`,...,`f_m`)`⟧= {<*fpos*→ *apos* > | $f_{fpos} = a_{apos}$}
e.g. PassThroughArgs⟦`f2.foo(x,f1)` *in* `method m(f1,f2)`⟧= {<1→3>,<2→1>}

Input: *SpecializationThreshold*, the minimum *Weight*(*arc*) for an arc to be considered for specialization
Output: $Specializations_m$: set of tuples of sets of classes for which each method *m* should be specialized

```
specializeProgram() =
  foreach method m do
      $Specializations_m$ := AppliesTo(m);
  foreach method m do
      if potentialSpecializationBenefit(m) > SpecializationThreshold then
          specializeMethod(m);

specializeMethod(m) =
  foreach arc such that Caller(arc) = m and isSpecializableArc(arc) do
      if Weight(arc) > SpecializationThreshold then
          addSpecialization(m, neededInfoForArc(arc, AppliesTo(Callee(arc))));

isSpecializableArc(arc) returns bool =
  return  PassThroughArgs⟦CallSite(arc)⟧ $\neq \emptyset$ and
          AppliesTo(Caller(arc)) $\neq$ neededInfoForArc(arc, AppliesTo(Callee(arc)));

neededInfoForArc(arc, calleeInfo) returns Tuple[Set[Class]] =
  needed := AppliesTo(Caller(arc));
  foreach <fpos→ apos> $\in$ PassThroughArgs⟦CallSite(arc)⟧ do
      $needed_{fpos}$ := $needed_{fpos} \cap calleeInfo_{apos}$;
  return needed;

addSpecialization(m, specTuple) =
  foreach existingSpec $\in$ $Specializations_{meth}$ do
      if specTuple $\cap$ existingSpec $\neq \emptyset$ then
          $Specializations_{meth}$ := $Specializations_{meth} \cup$ (specTuple $\cap$ existingSpec);
  foreach arc such that Callee(arc) = m do
      cascadeSpecializations(arc, spec);

cascadeSpecializations(arc, calleeSpec) =
  if PassThroughArgs⟦CallSite(arc)⟧ $\neq \emptyset$ and
          potentialSpecializationBenefit(Caller(arc)) > SpecializationThreshold and
          AppliesTo(Caller(arc))= neededInfoForArc(arc, AppliesTo(Callee(arc))) then
      callerSpec := neededInfoForArc(arc, calleeSpec);
      if callerSpec $\neq \emptyset$ and callerSpec $\notin$ $Specializations_{Caller(arc)}$ then
          addSpecialization(Caller(arc), callerSpec);
```

Figure 4.4: Selective specialization algorithm

occurs when the formal is passed directly as an actual parameter in the call; we call such a call site a *pass-through* call site (a similar notion is found in the jump functions of Grove and Torczon [Grove & Torczon 93]). The *PassThroughArgs* function provides a mapping from the method's formal parameters to the actual arguments used at the call site. A call site *c* is a pass-through call site if *PassThroughArgs*⟦c⟧ is non-empty. Our algorithm focuses on pass-through, dynamically-dispatched call sites, which we term *specializable call sites*, as the sites that can potentially benefit from specialization. For the `overlaps` example, the sends of ***`do`*** and ***`includes`*** are specializable call sites, since obtaining more precise information about the classes of `self` and `set2` can enable static binding of these message sends.

In summary, our algorithm visits each method in the call graph. The algorithm searches for high-weight, dynamically-dispatched, pass-through calls from the method, i.e., those call arcs that are both possible and profitable to optimize through specialization of the enclosing method. The algorithm computes the greatest subset of classes of the method's formals that would support static binding of the call arc, and creates a corresponding specialization of the method if such a subset of classes exists.

Much of the remainder of this section examines key aspects of the algorithm in more detail, focusing on the following issues:

- How is the set of classes that enable specialization of a call arc computed? This is computed by the *neededInfoForArc* function, as discussed in section 4.2.1.
- How should specializations for multiple call sites in the same method be combined? This is handled by the *addSpecialization* routine and is discussed in section 4.2.2.
- If a method *m* is specialized, how can we avoid converting statically-bound calls to *m* into dynamically-bound calls? Cascading specializations upwards (the *cascadeSpecializations* routine) can solve the problem in many cases, and is discussed in section 4.2.3.
- When is a method important to specialize? This is governed by the *potentialSpecializationBenefit*(*m*) function and the heuristics for determining the benefits of specializing a particular arc, and Section 4.2.4 discusses the tradeoffs involved.
- At run-time, how is the right specialized version chosen? This is discussed in section 4.2.5.

This section concludes with a discussion of issues related to the gathering and management of profile data, which is used to drive the specialization algorithm.

4.2.1 Computing Specializations for Call Sites

For each method that is deemed important to specialize, the algorithm visits each arc that might benefit from specialization and is of sufficient weight that it should be considered (see Section 4.2.4 for the heuristics involved). For each such arc, it determines the most general class set tuple for the pass-through formals that would allow static binding of the call arc to the callee method. This information is computed by the *neededInfoForArc* function, which maps the *AppliesTo* information for the callee routine back to the caller's formals using the mapping contained in the *PassThroughArgs* for the call site; if no specialization is possible, then the caller's *AppliesTo* information is returned unchanged. As an example, consider arc α from the call graph in Figure 4.3. For this arc, the caller's *AppliesTo* information is <{A,B,...,J},{A,B,...,J}>, the callee's *AppliesTo* tuple is <{B,E,H,I}>, and the *PassThroughArgs* mapping for the call site is {<2→1>}, so *neededInfoForArc*(α) is <{A,B,...,J}, {B,E,H,I}>. This means that within the specialized version, the possible classes of `arg2` are restricted to be in {B,E,H,I}; this information is sufficient to statically-bind the message send of `m2` to `B::m2()` within the specialized version. The *neededInfoForArc*(α) tuple will be added to the set of specializations for `m4`.

4.2.2 Combining Specializations for Distinct Call Sites

Different call arcs within a single method may generate different class set tuples for specialization. These different tuples need to be combined somehow into one or more specialization tuples for the method as a whole. Deciding how to combine specializations for different call sites in the same method is a difficult problem. Ideally, the combined method specialization(s) would cover the combinations of method arguments that are most common and that lead to the best specialization, but it is impossible, in general, to examine only the arc counts in the call graph and determine what argument tuples the enclosing method was called with. In our example, we can determine that within `m4`, the class of `self` was in {A,B,C,D,F} 625 times and in {E,H,I} 375 times, and that the class of `arg2` was in {B,E,H,I} 550 times and in {A,C,D,F,G,J} 450 times, but we cannot tell in what combinations the argument classes appeared.

Because we want to be sure to produce specializations that help the high-benefit call arcs, our algorithm errs on the side of producing method specializations for all plausible combinations of arc specializations that seem promising according to the profile data, even though this can produce method specializations that might not be used. In effect, we allow a limited degree of overspecialization in order to be sure that we don't underspecialize for a particular combination of arguments. Deciding how to combine a new specialization with existing specializations of the same method is the function of the *addSpecialization* function: given a new arc specialization tuple, it forms the combination of this new tuple with all previously-computed specialization tuples (including the initial unspecialized tuple) and adds these new tuples to the set of specializations for the method. For example, given two method specialization class set tuples $<A_1, \ldots, A_n>$ and $<A_1 \cap B_1, \ldots, A_n \cap B_n>$, adding a new arc specialization tuple $<C_1, \ldots, C_n>$ leads to four method specialization tuples for the method: $<A_1, \ldots, A_n>$, $<A_1 \cap B_1, \ldots, A_n \cap B_n>$, $<A_1 \cap C_1, \ldots, A_n \cap C_n>$, and $<A_1 \cap B_1 \cap C_1, \ldots, A_n \cap B_n \cap C_n>$ (assuming none of these intersections are empty; tuples containing empty class sets are dropped).

This approach can in principle produce a number of specializations for a particular method that is exponential in the number of specializable call arcs emanating from the method. As one way of limiting the combinations of specializations for a method, we currently only consider arcs whose dynamic weight is at least 5% of the total weight of the caller's specializable arcs, or whose inclusion does not add any new constraints on the set of specializations for the method. Even using this heuristic, it would in principle be possible to see a substantial number of specializations produced for a single method, but we have not observed this behavior in practice. For the Cecil benchmarks described in Section 3.4.1, we have observed an average of 1.9 specializations per method receiving any specializations, with a maximum of 8 specializations for one method. We suspect that in practice we do not observe exponential blow-up because most call sites have only one or two high-weight specializable call arcs and because methods tend to not have very many different formal arguments that are involved in heavily polymorphic message sends.

If exponential blow-up were to become a problem, the profile information could be extended to maintain a set of tuples of classes of the actual parameters passed to each method during the profiling run. Given a set of potential specializations, the set of actual tuples encountered during the profiling run could be used to see which of the specializations would actually be invoked with high frequency. However, it is likely to be more expensive to gather profiles of argument tuples than to gather simple call arc and count information.

4.2.3 Cascading Specializations

Before a method is specialized, some of its callers might have been able to statically-bind to the method. When specialized versions of a method are introduced, however, it is possible that the static information at these previously statically-bound call sites will not be sufficient to select the appropriate specialization. This is the case with the arc from `m3` to `m4` in the example: `m3` had static information that both its arguments were descendents of class A, which was sufficient to statically-bind to `m4` when there was only a single version of `m4`, but not after multiple versions of `m4` are produced.

In such cases, there are two choices: the statically-bound calls could be left unchanged, having them call the general-purpose version of the routine, or the statically-bound calls could be replaced with dynamically-bound calls that select the appropriate specialization at run-time. The right choice depends on the amount of optimization garnered through specialization of the callee relative to the increased cost of dynamically dispatching the call to the specialized method.

In some cases, this conversion of statically-bound calls into dynamically-dispatched sends can be avoided by specializing the calling routine to match the specialized callee method. This is the purpose of the *cascadeSpecializations* function. Given a specialization of a method, it attempts to specialize statically-bound pass-through callers of the method to provide the caller with sufficient information to statically-bind to the specialized version of the method. *cascadeSpecializations* first checks to make sure the call arc was statically bound (with respect to the pass-through arguments) and of high weight; if the call arc is dynamically bound, then regular specialization through *specializeMethod* will attempt to optimize that arc. If the calling arc passes this first test, *cascadeSpecializations* computes the class set tuple for which the caller should be specialized in order to support static binding of the call arc to the specialized version. If the algorithm determines that the call site can call the specialized version, and that specialization of the caller is necessary to enable static binding, the algorithm recursively specializes the caller method (if that specialization hasn't already been created). This recursive specialization can set off ripples of specialization running upwards through the call graph along statically-bound pass-through high-weight arcs. Recursive cycles of statically-bound pass-through arcs do not need special treatment, other than the check to see whether the desired specialization has already been created.

kind(arc): An arc's kind is an annotation on the profile-derived call graph, indicating what kind of calling sequence was used in making the call from caller to callee, chosen from the following definitions:

InlinedArc - arc was for a statically-bound, inlined call
StaticallyBoundArc - arc was for a statically-bound, but not inlined call
PredictedArc - arc was inlined, but as the result of a type prediction test inserted in caller
DynamicallyBoundArc - arc was from a dynamically-bound message send

ArcKindCost(*arc*) is a function that gives relative costs of different kinds of arcs (the absolute values are somewhat related to the number of instructions needed to implement a call of a particular kind of arc plus some approximation of the indirect cost imposed by the arc). For our implementation,
we used the following function:

```
ArcKindCost(arc) =   0,   if kind(arc) = InlinedArc
                     3,   if kind(arc) = StaticallyBoundArc
                     6,   if kind(arc) = PredictedArc
                     15,  if kind(arc) = DynamicallyBoundArc
```

ArcCost(*arc*) = *Weight*(*arc*) * *ArcKindCost*(*arc*)

```
potentialSpecializationBenefit(m) =
  benefit = 0;
  foreach arc such that Caller(arc) = m and isSpecializableArc(arc) do
      benefit = benefit + ArcCost(arc);
  return benefit;
```

Figure 4.5: Definition of *potentialSpecializationBenefit* function

4.2.4 Cost-Benefit Heuristics

Our implementation of the algorithm currently combines several heuristics in order to determine which methods are important to specialize and which call arcs within these methods methods should influence the specialization. The first heurustic, *potentialSpecialization-Benefit*(*m*), attempts to estimate the potential benefits of specializing a method *m*. It is illustrated in Figure 4.5 and works by finding all the outgoing arcs from the method m that specialization might benefit (i.e. that are not currently statically-bound and which are pass-through call sites). It then computes a potential specialization benefit from these arcs by summing their *ArcCost* values. The algorithm selects all those methods whose *potentialSpecializationBenefit* is larger than a minimum threshold for further examination. For these methods, the algorithm only considers those arcs whose *ArcCost* values account for more than 5% of the sum of the *ArcCosts* for a routine's specializable arcs. This is done was as a way of limiting the amount of specialization that is performed and, in particular, of limiting the number of disparate specializations that must be combined, as disucssed in Section 4.2.2.

After accumulating a set of specializations for a routine and estimating the benefits of each specialization, by summing the *ArcCost* heuristic for each arc that is expected to be resolved by specialization, the algorithm then reduces these benefit estimates by an estimate of how many upstream callers will be converted from statically-bound to dynamically-bound calls. Adding an additional argument position to the set of argument positions that are dispatched is also considered, and the benefits are reduced appropriately (since a more expensive dispatching mechanism must be used when more argument positions participate in the dispatching process). After these benefit calculations have been made, a final decision of whether or not to specialize is made, based on whether the remaining benefit is larger than a *specializationThreshold* value. In our implementation, we chose a value of 1000 for *specializationThreshold*. Some experimentation revealed that the performance of the algorithm was not very sensitive to the choice of this value.

These heuristics are clearly not perfect. First, code space increase incurred by specializing is not considered. In practice, however, we have not seen dramatic code space increases and therefore have not focused on including this information in the heuristics. Second, the hrustics treat all dynamic dispatches as equally costly. Due to the indirect effects of optimizations such as inlining, the benefits of statically binding some message sends can be much greater than others. A more sophisticated heuristic, such as that described in Chapter 6, could estimate the performance benefit of static binding and subsequent inlining, taking into account post-inlining optimizations.

In our system, the tradeoffs implemented by the heuristics seem to make reasonable decision in practice, but are clearly an area worthy of further exploration.

4.2.5 Selecting the Appropriate Specialized Method

At run-time, message lookup needs to select the appropriate specialized version of a method. In singly-dispatched languages like C++ and Smalltalk, existing message dispatch mechanisms work fine for selecting among methods that are specialized only on the receiver argument. Allowing specialization on arguments other than the receiver, however, can create *multi-methods* (methods requiring dispatching on multiple argument positions) in the implementation, even if the source language allows only singly-dispatched methods. If the runtime system does not already support multi-methods, it must be extended to support them. A number of efficient strategies for multi-method lookup have been devised, as discussed in

Section 2.3.5. The choice of a multi-method dispatching mechanism is orthogonal to our specialization algorithm, however.

4.2.6 Gathering and Managing Profile Information

Obtaining the profile data needed by the specialization algorithm requires that the program executable be built with the appropriate instrumentation. To enable long profiling runs and profiling of typical application usage, profiling should be as inexpensive as possible, since otherwise it may not be feasible to gather profile information. Computing counts for statically-bound arcs can be done by inserting counters at statically-bound call sites. The expense of profiling dynamically-dispatched message sends depends in large part on the run-time system's message dispatching mechanism. Some systems, including the Vortex compiler's implementation of the Cecil language in which we implemented the algorithm, use *polymorphic inline caches* [Hölzle et al. 91]: call-site-specific association lists mapping receiver classes to target methods (see Section 2.3.5). To gather call-site-specific profile data, counter increments are added to each of the cases, and the counters for the PICs of all call sites are dumped to a file at the end of a program run. The run-time overhead of this profiling for the Cecil benchmark programs is 15-50%. In other systems that use method dispatching tables, such as C++ and Modula-3 implementations, additional code is inserted to maintain profile information at message send sites.

To make managing profile information more convenient, Vortex maintains a persistent internal database of profile information that is consulted transparently during compilations. Additionally, for a suite of object-oriented programs written in Self, Cecil, and C++, we have observed that the kind of profile information needed to construct this call graph remains fairly stable across different inputs to a program and even as the program evolves [Grove et al. 95], so there is some evidence that profiling can be done relatively infrequently and reused across many compilations.

4.3 Performance Evaluation

To evaluate the performance of the algorithm, we used a benchmark suite consisting of the three largest Cecil programs described in Section 3.4.1: scheduler, typechecker, and compiler. (Since the C++ and Modula-3 front-ends for Vortex do not expose their method dispatching tables at a high enough level, we are unable to experiment with specialization for these languages). We performed a number of experiments with the goal of answering several questions:

- How much does selective specialization improve performance when it is added to the best static analysis system described in Section 3.4, which performs class hierarchy analysis and exhaustive testing?
- How does customization compare with whole-program optimization techniques such as class hierarchy analysis, or class hierarchy analysis augmented with exhaustive class-hierarchy-guided class prediction?

We examined the following compiler configurations, both with and without profile-guided class prediction[2]:

- **intra** (or **i**) - The intra configuration from Section 3.4, which performs intraprocedural class analysis and hardwired class prediction for common messages, but compiles a single routine for each source method.
- **i+cust** - The intra configuration plus customization, which compiles a new version of a method for each different class that inherits the method.
- **i+CHA+exh** - The same configuration as described in Section 3.4, which performs intraprocedural class analysis, class hierarchy analysis, and exhaustive class testing.
- **i+CHA+exh+spec** - The i+CHA+exh configuration augmented with the selective specialization algorithm. Class hierarchy analysis is included because the selective specialization algorithm requires it in order to compute the *AppliesTo* tuples.

To evaluate the performance of the algorithm, we measured both abstract qualities, such as the dynamic number of dynamic dispatches and class and cone tests in each version of the programs, as well as absolute criterion such as bottom-line execution speed and compiled code space. The results are shown in Table 4.1.

All of the configurations show large performance improvements (often approaching a factor of two) over a system that performs only intraprocedural class analysis, since they all provide substantially more static class information to the optimizer than simple intraprocedural

2. The kinds of profile information required by the specialization algorithm (`gprof`-style weights of call graph arcs) is different from the kind of information required by profile-guided class prediction (tuples of common classes at each call site), so it makes sense to evaluate selective specialization both with and without profile-guided class prediction.

Table 4.1: Impact of specialization and customization

Program		Configuration	Execution time (secs)	Message sends executed	Compiled code space (bytes)	Class tests + cone tests
Scheduler		intra	9.69 (1.00)	5,662,957 (1.00)	1,936,032 (1.00)	2,306,070 (1.00)
		i+cust	5.26 (0.54)	1,566,193 (0.28)	4,378,376 (2.26)	2,443,193 (1.06)
		i+CHA	6.58 (0.68)	2,577,346 (0.46)	1,826,712 (0.94)	2,031,323 (0.88)
		i+CHA+exh	5.32 (0.55)	1,599,429 (0.28)	1,956,088 (1.01)	2,506,140 (1.09)
		i+CHA+exh+spec	4.98 (0.51)	1,401,864 (0.25)	2,029,040 (1.05)	2,271,423 (0.98)
	W/Profile	intra	3.11 (0.32)	466,123 (0.08)	2,047,688 (1.06)	2,296,420 (1.00)
		i+cust	3.41 (0.35)	593,122 (0.10)	4,745,104 (2.45)	2,292,520 (0.99)
		i+CHA	2.61 (0.27)	398,716 (0.07)	1,921,064 (0.99)	2,008,608 (0.87)
		i+CHA+exh	2.74 (0.28)	377,142 (0.07)	2,008,728 (1.04)	2,215,137 (0.96)
		i+CHA+exh+spec	2.76 (0.28)	361,968 (0.06)	2,081,200 (1.07)	2,152,962 (0.93)
Typechecker		intra	103.44 (1.00)	62,357,301 (1.00)	4,519,096 (1.00)	18,238,949 (1.00)
		i+cust	57.15 (0.55)	16,754,599 (0.27)	10,840,416 (2.40)	20,803,737 (1.14)
		i+CHA	64.79 (0.63)	22,382,240 (0.36)	4,159,944 (0.92)	15,768,735 (0.86)
		i+CHA+exh	50.55 (0.49)	13,712,846 (0.22)	4,440,144 (0.98)	20,953,271 (1.15)
		i+CHA+exh+spec	46.49 (0.45)	11,031,174 (0.18)	4,594,656 (1.02)	22,698,360 (1.24)
	W/Profile	intra	33.84 (0.33)	6,961,025 (0.11)	5,090,344 (1.13)	28,736,043 (1.58)
		i+cust	37.12 (0.36)	6,885,251 (0.11)	11,918,720 (2.64)	23,876,903 (1.31)
		i+CHA	30.08 (0.29)	6,409,824 (0.10)	4,578,232 (1.01)	20,360,226 (1.12)
		i+CHA+exh	29.78 (0.29)	5,605,325 (0.09)	4,735,416 (1.05)	21,631,091 (1.19)
		i+CHA+exh+spec	29.87 (0.29)	5,521,203 (0.09)	4,879,536 (1.08)	21,429,207 (1.17)
Compiler		intra	1,500 (1.00)	617,004,841 (1.00)	9,754,032 (1.00)	158,966,133 (1.00)
		i+cust	772 (0.51)	169,298,302 (0.27)	31,119,400 (3.19)	202,501,598 (1.27)
		i+CHA	903 (0.60)	246,662,311 (0.40)	9,825,936 (1.01)	132,188,864 (0.83)
		i+CHA+exh	700 (0.47)	153,558,538 (0.25)	9,985,384 (1.02)	198,647,910 (1.25)
		i+CHA+exh+spec	639 (0.43)	129,811,734 (0.21)	10,302,752 (1.06)	179,658,892 (1.13)
	W/Profile	intra	615 (0.41)	124,908,520 (0.20)	10,708,512 (1.10)	273,418,374 (1.72)
		i+cust	532 (0.35)	79,130,611 (0.13)	31,200,736 (3.20)	202,074,062 (1.27)
		i+CHA	515 (0.34)	90,153,322 (0.15)	9,584,728 (0.98)	175,309,100 (1.10)
		i+CHA+exh	486 (0.32)	71,685,978 (0.12)	10,343,920 (1.06)	201,732,501 (1.27)
		i+CHA+exh+spec	506 (0.34)	71,587,088 (0.12)	10,596,760 (1.09)	196,963,036 (1.24)

class analysis. Comparing the various configurations without profile-guided class prediction, we find that customization performs about 15% better than a system that performs class hierarchy analysis alone. Adding exhaustive testing to class hierarchy analysis tilts the balance the other way, performing about 10% better than a system that performs customization. Adding exhaustive testing to a system that performs customization is not a useful configuration, since exhaustive testing requires class hierarchy analysis and customization is generally used only in systems that do not examine the entire class hierarchy, as a means of recovering some information in systems that have “open world” assumptions.

Selective specialization improves performance by 8-10% over a system that performs class hierarchy analysis and exhaustive testing, and performs about 5-20% better than a system that performs customization. All of the other configurations have substantially smaller space costs than do the customization configurations, which often have executables that are three times larger than the other configurations (with commensurately longer compile times). All of the configurations had much smaller effects when combined with profile-guided class prediction, since profile-guided class prediction was already predicting for and inlining many message sends at the relatively low cost of a class test.

Each of these optimization techniques has advantages and disadvantages. The primary disadvantages of customization are the substantial compiled code space and compile time costs: the executables for programs compiled with customization ranged from 126% to 220% larger, and compilation times increased by roughly the same factor. The space cost and compile-time cost of customization also increase more for larger programs with more classes, deeper inheritance hierarchies, and more methods, making customization increasingly impractical for realistically-sized programs. In comparison, selective specialization increases code space by less than 10% over the intra configuration and by less than 4% over the i+CHA+exh configuration. The primary disadvantages of the selective specialization algorithm are that it requires that the entire class hierarchy of the program be available at compile time, precluding separate compilation, and that it requires gathering the profile data needed to derive the weighted call graph.

In practice, the kinds of methods that were chosen for specialization by the specialization algorithm seemed to be similar to the kinds of routines presented in the example code in Section 4.1: frequently-used, factored methods that had relatively large bodies and that had many message sends to self or other arguments in their bodies, where the targets of these message sends were methods implemented in subclasses below the point where the factored routine was located. These kinds of methods are not optimized very effectively by class hierarchy analysis (or by exhaustive class testing when there are a large number of candidate methods for the contained message sends), and the specialization algorithm effectively unfactors the factored routine to the points in the hierarchy where there will be sufficient information to statically-bind the high frequency message sends to the argument classes.

4.4 Related Work

The implementations of Self [Chambers & Ungar 91], Trellis [Kilian 88], and Sather [Lim & Stolcke 91] use customization to provide the compiler with additional information about the class of the receiver argument to a method, allowing many message sends within each customized version of the method to be statically bound. All of this previous work takes the approach of always specializing on the exact class of the receiver and not specializing on any other arguments. The Trellis compiler merges specializations after compilation to save code space, if two specializations produced identical compiled code; compile time is not reduced, however. As discussed in section 4.1, customization can lead to both overspecialization and underspecialization. Our approach is more effective because it identifies sets of receiver classes that enable static binding of messages and uses profile data to ignore infrequently-executed methods (thereby avoiding overspecialization), and because it allows specialization on arguments other than just the receiver of a message (preventing underspecialization for arguments).

Cooper, Hall, and Kennedy present a general framework for identifying when creating multiple, specialized copies of a procedure can provide additional information for solving interprocedural dataflow optimization problems [Cooper et al. 92]. Their approach begins with a conservative approximation to the program’s call graph. It first makes a forward pass over the call graph propagating “cloning vectors” which represent the information available at call sites that is deemed interesting by the called routine, and then makes a second pass merging cloning vectors that lead to the same optimizations. The resulting equivalence classes of cloning vectors indicate the specializations that should be created. Their framework applies to any forward dataflow analysis problem, such as constant propagation. Our work differs from their approach in several important respects. First, we do not assume the existence of a conservative call graph prior to analysis. Instead, we use a subset of the program’s call graph derived from dynamic profile information. This is important, because an accurate call graph is difficult to compute when dynamically-dispatched messages are used extensively (see Section 2.3.1). Second, our algorithm is tailored to object-oriented languages, where the information of interest is derived from the specializations of the arguments of the called routines. Our algorithm consequently works backwards from dynamically-dispatched pass-through call sites, the places that demand the most precise information, rather than proceeding in two phases as does Cooper *et al.*'s algorithm. Finally, our algorithm exploits profile information to select only profitable specializations. On the other hand, because the call graph used in our algorithm is not conservative, the assumptions and transformations made in our specialization algorithm must always preserve correctness in the case of the program exercising a path through the call graph that was not in the input call graph provided to the specialization algorithm.

Procedure specialization has long been incorporated as a principal technique in partial evaluation systems [Jones et al. 93]. Ruf, Katz, and Weise [Ruf & Weise 91, Katz & Weise 92] address the problem of avoiding overspecialization in their FUSE partial evaluator. Their work seeks to identify when two specializations generate the same code. Ruf identifies the subset of information about arguments used during specialization and reuses the specialization for other call sites that share the same abstract static information. Katz extends this work by noting when not all of the information conveyed by a routine's result is used by the rest of the program. Our work differs from these in that we are working with a richer data and language model than a functional subset of Scheme and our algorithm exploits dynamic profile information to avoid specializations whose benefits would be accrued only infrequently.

4.5 Summary

This chapter has presented a general framework for specialization in object-oriented languages and an algorithm that combines static analysis and profile information to identify the most profitable specializations. As demonstrated in Section 4.3, the algorithm is especially effective at providing performance improvement with very modest space increases (on the order of 5%). In contrast, previous techniques such as customization offer little or no performance advantage over a compiler that combines selective specialization with class hierarchy analysis, yet increase code space by *factors* of 3 or 4, and become largely impractical for large programs for deep inheritance hierarchies. Furthermore, because of its more judicious application of specialization, our specialization algorithm is appropriate for specializing on multiple arguments of a method and for use in object-oriented languages with multi-methods.

Chapter 5

Selective Recompilation

With a system that supports true separate compilation, changes to the implementation of one module can never require the regeneration of code for a different module. However, a compiler that performs optimizations across module boundaries gives up this property, making the generated code for one module dependent on various aspects of the source code from other modules. For example, inlining a routine across a module boundary makes the generated code of one module dependent on the source code of the body of the routine that was inlined. Other interprocedural optimizations, such as class hierarchy analysis, can introduce more subtle kinds of dependencies. One approach for ensuring the correspondance between source and object modules is to always recompile all modules in the system after each programming change. This approach might be acceptable in an environment where interprocedural optimizations are used only as a final, batch-style optimization and the overhead of this full recompilation is incurred only rarely. However, supporting cross-module optimizations in an environment that strives for fast turnaround after programming changes requires that the compiler be able to selectively regenerate only those pieces of derived information (e.g., interprocedural summaries or generated code) that are affected by a programming change. To provide this incrementality, the compiler needs some sort of dependency mechanism that links source modules (or pieces of modules) to information derived from the source modules is needed.

Ideally, the dependency mechanism should be *concise* (use little storage space) and *speedy* (take little time to update the derived information). Speediness is dependent on *rapid* and *selective* calculation of out-of-date derived information: we would like a dependency mechanism that quickly identifies the information that needs to be recomputed and does not recompute more than necessary. Different tradeoffs can be made among conciseness, rapid invalidation, and selectiveness when engineering a cost-effective dependency structure. For example, the UNIX make utility [Feldman 79] maintains a coarse-grained dependency structure at the file-level granularity, which is space-efficient but not very selective. Alternatively, a finer-grained dependency structure can be maintained that is selective but which consumes more space. To support the design and implementation of dependency systems for compilers that perform cross-module optimizations, we have developed a framework for the representation of dependency information. Section 5.1 provides a brief description of this framework and Section 5.3 focuses on how it used in the Vortex compiler to maintain fine-grained, selective dependencies in the presence of the optimization techniques described in Chapter 3. Further details about the framework can be found elsewhere [Chambers et al. 95].

5.1 Dependency Framework

In our framework, intermodule dependencies are represented by a directed, acyclic graph structure. Nodes in this graph represent information, and an edge from one node to another indicates that the information represented by the target node is derived from or depends on the information represented by the source node. Based on the number of incoming and outgoing edges, we classify nodes into three categories, shown in Figure 5.1.

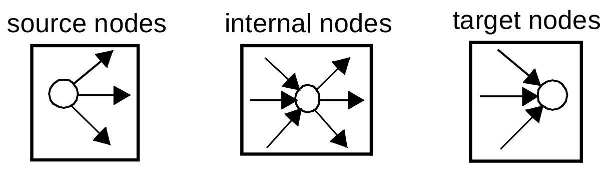


Figure 5.1: Kinds of dependency nodes

- *Source nodes* have only outgoing dependency edges. They represent information present in the source modules comprising the program, such as the source code of procedures and the class inheritance hierarchy.
- *Target nodes* have only incoming dependency edges. They represent information that is an end product of the compiler, such as compiled code.
- *Internal nodes* have both incoming and outgoing edges. They represent information that is derived from some earlier information and used in the derivation of some later information. Interprocedural summary information and the procedure call graph are examples of information that could be represented by internal nodes. The information represented by an internal node may be both an end product (such as a call graph for a

call graph browser) and an input for other information (such as a call graph used during interprocedural optimizations).

Filtering nodes are a special kind of internal node in our framework, and they are especially useful in implementing selective dependency mechanisms. In many cases, the mere fact that the source code of the program has changed does not necessarily imply that all target nodes that are downstream of the source node in the dependency graph are out of date, since the dependence might be on an aspect of the source node that was unaffected by the programming change. For example, a common kind of interprocedural summary information is a set of all global variables modified by each routine. If a procedure's body is changed, this set of defined globals might have changed, but it's also possible that the set might be unaffected by the source change. We wish to avoid assuming that any users of the interprocedural summary information are out of date, just because a procedure has been edited, i.e., we wish to be more selective in our invalidations. To support this, filtering nodes include an additional predicate that is tested whenever the filtering node is invalidated. Only if the predicate test fails are the downstream dependencies invalidated. In this example, the filtering node would first recompute the set of defined globals and compare it to the old set. Only if the sets differed would the invalidation be propagated to downstream nodes, ensuring that invalidations will propagate only as far as derived information really has changed.

The dependency graph is constructed incrementally as information is computed. Whenever the compiler uses a piece of information that might change, it adds an edge to the dependency graph from the node representing the used information to the node representing the client. In our implementation, there is a notion of the "current dependency node" representing the active client. Whenever a piece of information is queried that might change, the provider of the information automatically adds a link from its representative node to the current dependency node. In this manner, construction of the dependency graph is largely transparent to client code. When performing computation that might affect an internal node in the dependency graph, the active client explicitly switches thecurrent dependency node to the internal node. It then performs the computation, potentially adding new links from sources of information to its internal dependency node (which is currently the active dependency node). After finishing its computation, it restores the old value of the current dependency node, and often will add a link to it from its internal dependency node, if appropriate.

Invalidation proceeds by propagating invalidate messages through the dependency graph. The compiler front-end detects what pieces of the source program have changed as a result of a programming change. The corresponding source nodes are sent the invalidate message. In the simplest case, when a node receives an invalidate message, it updates the information it represents and then resends invalidate to each of its successors, if any, in the dependency graph. In this manner, all information derived from out-of-date source code will be updated. This process can be somewhat complicated by the restriction that an internal node in the dependency graph can't recompute its information until the information represented by its predecessors in the graph has been recomputed (since it might otherwise use old, stale information to recompute its new value). To handle this, the framework ensures that invalidations are propagated through the graph in a topological manner. Further details about how this is done efficiently can be found elsewhere [Chambers et al. 95].

5.2 Related Work

Selective recompilation attempts to minimize the amount of code that must be recompiled to maintain the correctness of the generated code with respect to a program's source code. Much previous work proposes ways of insulating compiled code from source changes through the use of more selective dependency mechanisms. Adams *et al.* provides an overview of several of these techniques and a quantitative comparison of their effectiveness in reducing compilation time [Adams et al. 94]. In this section, we briefly review this previous work and illustrate how our framework can be applied to model several of these strategies, using as an example a specification for a module A (A_{spec}) that declares a procedure p, a structure s, and a global variable v, and implementations for modules A (A_{impl}) and B (B_{impl}) that both import A_{spec}.

The simplest selective recompilation rule, sometimes referred to as *cascading recompilation* and exemplified by the UNIX `make` utility, is that whenever a module specification changes, all module implementations that import that specification must be recompiled. Under cascading recompilation, both A_{impl} and B_{impl} would be recompiled whenever A_{spec} changes, a situation modeled with the following simple dependency graph shown in Figure 5.2.

Smart recompilation [Tichy & Baker 85] improves selectivity by maintaining dependencies on individual declarations within a specification, rather than on whole specifications.[1]

1. In *smartest recompilation* [Shao & Appel 93], modules are compiled independently to avoid creating intermodule dependencies, and so smartest recompilation has no need of an intermodule dependency mechanism.

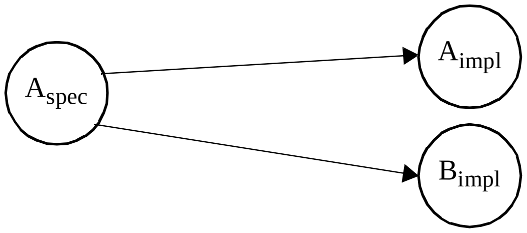


Figure 5.2: `make`-like dependency structure

Since modules frequently utilize only a subset of the declarations in an imported specification, recompilation is avoided when declarations outside of this subset are changed. Suppose that module A uses both procedure `P` and structure `S`, but that module B uses only structure `S`. A reasonable application of our framework would construct a fine-grained dependency structure, where root nodes are associated with individual declarations in a specification module, as shown in Figure 5.3.

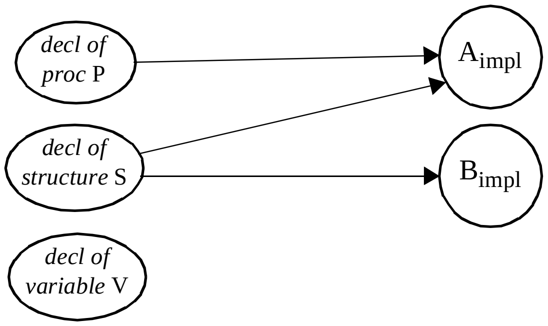


Figure 5.3: *Smart recompilation* dependency structure

This framework would require more space to represent than the basic framework for cascading recompilation, but it supports much better selectivity. Moreover, since it is likely that many implementations will depend on the same groups of declarations, automatic factoring nodes inserted by our framework can help reduce the space cost for the fine-grained dependency graph.

Attribute recompilation [Dausmann 85] extends smart recompilation to allow dependencies to depend on particular attributes of declarations, rather than depending on the whole declaration. For example, suppose B_{impl} depends only on the size of structure `S`, not on the individual elements of the structure. This can be modelled in our framework with a filtering dependency node that propagates invalidations only when some attribute of the declaration changes, in this case when the size of structure S changes, as shown in Figure 5.4

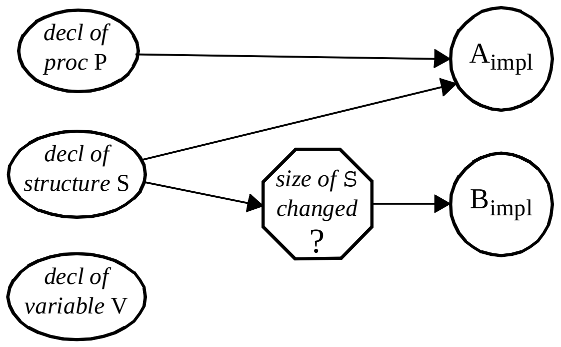


Figure 5.4: *Attribute recompilation* dependency structure

Using our dependency framework, these dependency mechanisms become simple to express and implement.

Compilers that perform interprocedural optimizations generate quite subtle intermodule compilation dependencies. Examples of such optimizations include cross-module inlining and using the results of interprocedural summary information, such as available constants or aliasing relationships, from one module to justify optimizations performed in another module. Burke and Torczon describe several schemes for performing selective recompilation in the face of common interprocedural optimizations for procedural languages [Burke & Torczon 93]. Under their compilation model, each procedure has several associated interprocedural summary sets. The simplest form of selective recompilation is to recompile a procedure whenever the body of the procedure changes or when an interprocedural set used during the compilation of the procedure changes. This can easily be modelled by creating dependency graph edges from interprocedural summary sets to the compiled code for the procedures that used the summary sets. Since the compilation of a procedure P need not depend on the exact contents of the summary set, but often only some property of the summary set, a filtering node mediating between a summary set and a compiled procedure can model more precisely the part of the summary set on which the compilation really depended, as shown in Figure 5.5.

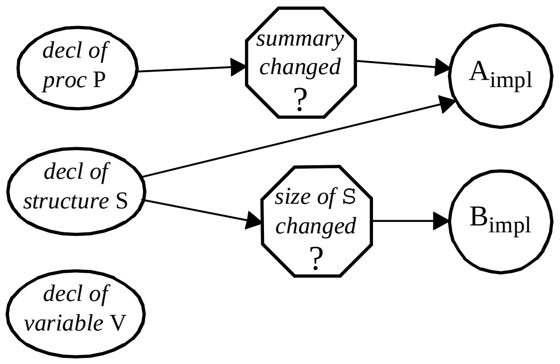


Figure 5.5: Dependencies for interprocedural optimizations

The system described by Burke and Torczon performs a global pass after each programming change in which interprocedural summary information is recomputed from scratch, after which their recompilation tests are used to determine what procedures need to be recompiled as a result of changes in the interprocedural information. The FIAT system improves on this by computing interprocedural summary information lazily [Hall et al. 93]. Using our dependency framework, however, the computation and update of the interprocedural summary information could be made incremental, with the framework only invalidating and recomputing those summary sets that are out of date.

5.3 Vortex Dependency Graph Structure

Interprocedural optimizations for object-oriented languages, such as class hierarchy analysis, provide a challenging environment for implementing selective recompilation. In part this is due to the large variety of source changes that can occur in programs that include structures such as class hierarchies and method definitions. As an additional challenge, many optimizations are unaffected by many kinds of source changes, so it is essential to make heavy use of filtering nodes to quash spurious invalidations eminating from ineffectual source changes. This section describes the way in which the framework for dependency information described in Section 5.1 is used to support incremental recompilation in the face of the interprocedural optimizations performed by Vortex. The structure of a large portion of the Vortex compiler's dependence mechanism is shown in Figure 5.6. Each node in the figure stands for all the

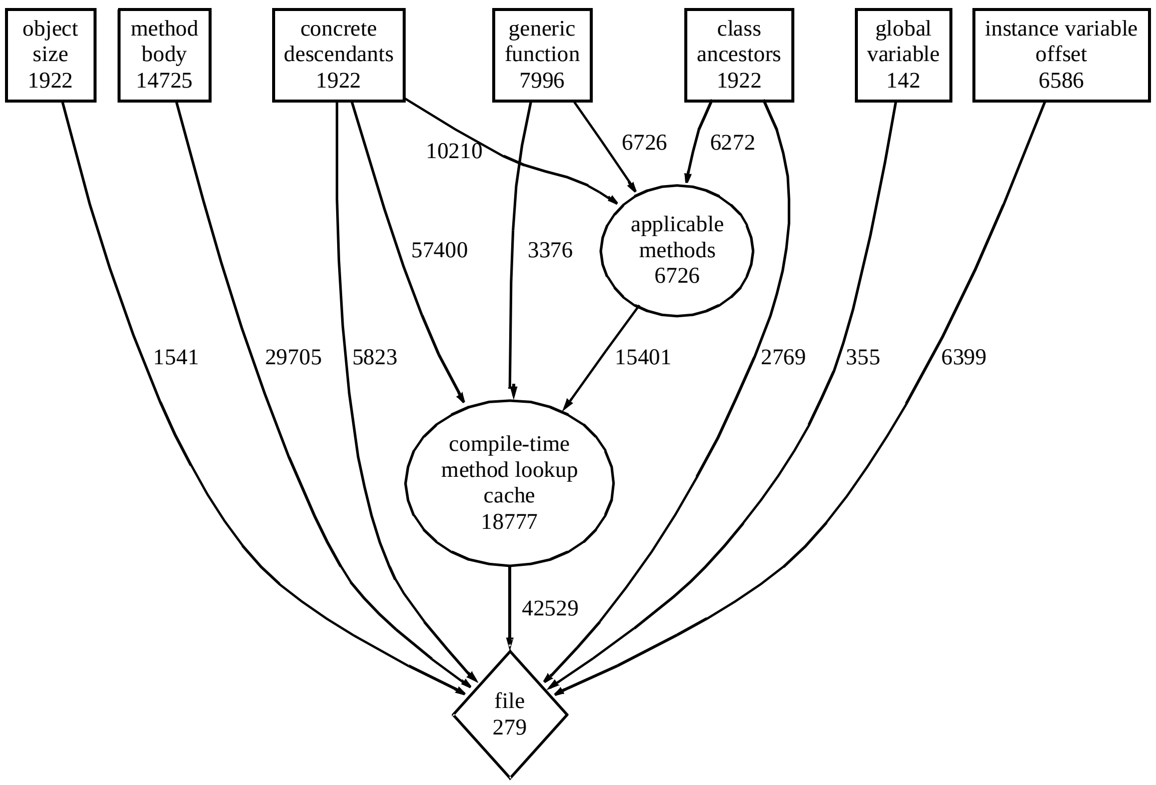


Figure 5.6: Vortex Dependency Graph Structure

dependency nodes of a particular type, and an edge between two nodes in the figure represents all the edges in the dependency graph from a dependency node of the first type to a dependency node of the second type. The numbers in the figure indicate how many nodes and edges of a particular type are present in the dependency graph after compiling the Vortex compiler from scratch with full optimization. The method lookup cache is the focal point of the dependence mechanism, and is essentially a table mapping a message name and a tuple of static information known about the arguments to a set of methods that might be invoked given this information. Each entry in the cache has a filtering dependency node associated with it, that depends (directly or indirectly) on all parts of the source program that might affect the outcome of the method lookup. If any of these properties change, the filtering node re-executes the method lookup. Only if the outcome of this lookup changes, however, is any dependent compiled code invalidated and recompiled. By interposing this filtering node between the generated code and the source dependencies, we gain selectivity and insensitivity to superficial source changes that do not affect the outcome of method lookups.

Table 5.1 provides a brief summary of what the various kinds of dependency nodes represent, how they get introduced into the graph, and some notes particular to the way in which we use each kind of dependency node in our environment.

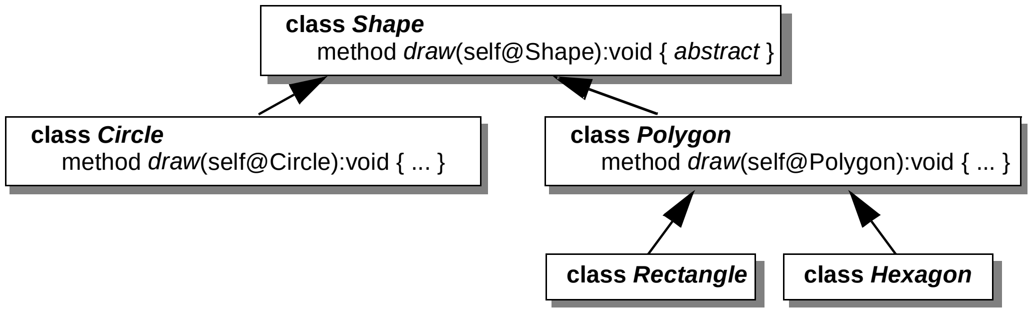
Table 5.1: Kinds of dependence nodes and their causes

Variable dependence	**Represents**: A dependence on the presence of a particular global variable, and on its representation	**Introduced by:** Referencing the variable	
Instance variable offset dependence	**Represents**: A dependence that a particular instance variable is at a particular offset in its containing object	**Introduced by:** Referencing the instance variable	**Notes:** In our environment, we always add new instance variables onto the end of the class and the ends of all subclasses, to avoid changing the offsets of other instance variables unnecessarily. When an instance variable is deleted, however, we do move all subsequent instance variables up, invalidating all code that was dependent on their offsets. The class layout algorithm in the compiler could leave padding in place of the deleted instance variable to avoid this, with the number of outgoing edges from instance variable dependencies providing a heuristic of the recompilation cost of moving an instance variable, or it could move a single instance variable into the freed up instance variable offset, rather than shifting the offsets of all subsequent instance variables down by one (which can cause many more invalidations).
Object size dependence	**Represents**: A dependence on the size of a particular object	**Introduced by:** Compiling code that needs the size of the object, such as creating an instance of the object	
Method body dependence	**Represents**: A dependence on source code of method body	**Introduced by:** Inlining the method	**Notes:** To avoid spurious invalidations, the system first does a quick textual comparison of the old and new method body. If this fails, it does a more expensive check of comparing the token streams generated by scanning the old and new method bodies. This prevents superficial changes such as editing comments or changing indentation within the method from invalidating code.

Table 5.1: Kinds of dependence nodes and their causes

Class ancestors dependence **Represents**: A dependence on the set of ancestors of a particular class **Introduced by:** Comparing two classes to determine their relationship (relationship might change if either class's ancestors change) **Notes:** An earlier scheme introduced a dependence on the set of all *descendants* of each class when determining two classes' relationship. However, we found that the set of descendants of a class is much more volatile than its set of ancestors, and switching to ancestor dependencies eliminated many spurious invalidations.
Generic function dependence **Represents**: A dependence on the set of all methods in the program that share the same message name and number of arguments **Introduced by:** Internal dependence nodes that want to be notified when the set of methods making up a particular generic function has changed
Applicable methods dependence **Represents**: A dependence on the computation of the *appliesTo* information for a particular generic function. This computation is described in Chapter 3. **Introduced by:** Performing compile-time method lookup of a particular message introduces a dependence on the applicable methods dependence for that generic function, to ensure that the results of the lookup are recomputed if the *appliesTo* information for the generic function changes.
Compile-time method lookup cache entry dependence **Represents**: A dependence on the compile-time method lookup for a particular *message* × *static information* combination **Introduced by:** Using the result of compile-time method lookup to optimize a message send. **Notes:** Clients first perform the lookup without introducing dependencies. Only if they are able to optimize a message send using the results of the lookup do they introduce a dependency on the lookup cache node. In addition to providing a location for recompilation dependencies, this structure also serves as a cache to speed up the compilation process (since future call sites with the same *message* × *static information* will obtain the cached lookup results).

To see how the dependency graph gets built as a byproduct of performing optimizations, we will consider the Shape class hierarchy and call site shown in Figure 5.7, in which the draw message is sent to an instance that is known statically to be an instance of a subclass of the Polygon class. Using class hierarchy analysis, the compiler can replace the dynamically-dispatched draw message with a direct call to method draw(@Polygon), which can then be optimized further, such as by inlining. The correctness of this optimization depends on several things, including the existence of method draw(@Polygon), the set of subclasses of Polygon, and the lack of a draw method defined on any of these subclasses. For example, if a draw


class *Shape*
method *draw*(self@Shape):void { *abstract* }
class *Circle*
method *draw*(self@Circle):void { ... }
class *Polygon*
method *draw*(self@Polygon):void { ... }
class *Rectangle*
class *Hexagon*

```
let p:Polygon := ...;
p.draw;
```

Figure 5.7: Example class hierarchy and code fragment

method is added to Rectangle, or if a Triangle subclass with its own draw method is added, the optimizations of the draw call site will be incorrect. On the other hand, adding a new Triangle subclass that does not have its own draw method does not affect the correctness of this optimization of the draw call site. A dependency mechanism must be able to detect those changes that affect the correctness of derived information, while simultaneously striving to ignore other changes, in order to minimize the amount of recompilation performed.

The Vortex compiler creates the appropriate dependencies as a byproduct of computing the result of the method lookup at compile time. To see how this is done, we'll step through an example of optimizing the draw call site. At the point where this call site is first encountered, the current dependence is set to the dependence object representing the object file that is currently being compiled. The first step for optimizing this message send is to perform compile-time method lookup, by querying thecompile-time method lookup cache. To avoid introducing dependencies before it is known whether there is even sufficient information to perform an optimization, most of the information provides in the compiler that have associated dependencies support two interfaces: a no_dep version, which simply computes the requested information but does not create a dependency from the provider of the information to the current dependency node, and a normal version, which computes the information and also adds this dependency link. The usual mode of operations is for clients to first use the no_dep version to ascertain whatever information is needed to decide whether or not to perform the optimization, and then, if the optimization is to be performed, to requery the information source(s) using the normal interface, which adds the appropriate dependency links from the provider(s) of the information to the current dependency node. Initially, the optimizer queries the lookup cache using the lookup_no_dep operation, to see what possible methods could be invoked by sending draw to an object whose class is known to be Cone(Polygon). Assuming that such a query had not been made before, the lookup cache must determine the answer based on the structure of the source code, and dependencies must be introduced that force the reevaluation of the lookup if aspects of the source code material to the lookup change. Doing so requires several steps:

1. Switch the *current dependence* temporarily to the dependence node for a new method lookup cache node, which will hold the result of the method lookup.
2. Determine which draw methods are defined in the program. This operation creates a dependence on the draw generic function, so as to force a reevaluation of the method lookup cached by this cache entry when draw methods are added to or removed from the program.
3. Compute the appliesTo information for the draw methods (see Section 3.2.2). Computing this information involves determining the relationships between several different classes in the system (e.g. between Polygon, Circle, and Shape), and dependencies are introduced from the sets of ancestors of these classes to the *current dependence*, so that if ancestors are added to or deleted from these classes (which could potentially affect the relationships between the classes and thus the appliesTo information), then downstream information will be reevaluated.
4. Determine which of the methods in the draw generic function could be invoked, by intersecting the appliesTo information computed for each method with the static information supplied to the lookup, and determining where there is overlap. This can introduce dependencies from the set of descendants of a class to the *current dependence*, since these overlaps operations often convert static information to a bitset representation for ease of comparison (see Section 3.2.2).
5. Restore the old value of the *current dependence*.

The set of applicable methods that could be invoked from a call site sending the draw message with a particular amount of static information about its parguments is cached in the method lookup cache, so that future lookups can simply return the cached information, rather than recomputing it and recreating the appropriate dependency links. The result of the lookup is then returned to the client, which determines what optimizations can be performed. If the client decides to perform optimizations that depend on the set of methods returned, then a dependence is introduced from the lookup cache entry to the old value of *current dependence* (typically representing the compiled code for the object module where the optimization is performed).

Other kinds of optimizations are handled in a similar manner, through the use of internal caches, and interfaces that automatically introduce the appropriate dependencies whenever queries are made whose answers depend on aspects of the program's source code that might change.

5.4 Experiments

We performed several experiments to assess the selectivity and space costs of our dependency mechanism, relative to several other possible dependency mechanisms. This section reports on our methodology and on the results of our experiments.

5.4.1 Methodology

We have implemented our dependency framework in the context of the Vortex compiler and programming environment. Since we use make use of the dependency mechanism during our everyday development of the Vortex program, Vortex itself serves as a reasonably large and rapidly developing benchmark application for evaluating our dependency mechanism. At the time of these experiments, Vortex was about 40,000 lines of code. To perform the experiments, we modified the compiler to log source changes each time it was invoked; by replaying these logs we are able to recreate the original sequence of source modifications and monitor the subsequent recompilations with a variety of dependency mechanisms. The logs used to drive the simulations in this section cover a period of several weeks, during which 86 compilations occurred. The changes that occurred during this period are summarized in Table 5.2.

Utilizing this history of changes, we measured the performance of the following dependency systems for our system when dependencies were introduced by performing class hierarchy analysis and extensive cross-module inlining:

- Header per File evaluates a system where a single header file is simulated for each Cecil source file (Cecil does not require header files). This is meant to simulate a system built upon a standard `make` and header file-based system, as is common in C++ environments. Unfortunately, to support the kind of whole-program optimizations per-

Table 5.2: Summary of Source Changes over 86 Compiles

Program Structure	Initial	Changed	Added	Removed	Final
Class Declarations	903	0	12	42	873
Methods	7272	714	455	341	7386
Instance Variables	889	2	34	23	900
Global Variables	73	8	21	3	91
Lines	39,697	n/a	n/a	n/a	40,874

formed for draw, an implementation file must include (and be dependent on) the header files for all *subclasses* as well as superclasses of classes used in the implementation file. Whenever any of these header files is changed, the implementation file is recompiled. While simple and space-efficient, this approach is likely to have poor selectivity; our experimental results support this conclusion.

• Header per Class refers to a system where each class is assumed to have its own header file. We simulate this configuration because many of our Cecil source files define multiple classes, and we wanted to separate out the effects of these multiple definitions from the simple requirement of including an interface for each subclass of a class used in an implementation file.

• Self-like refers to a fine-grained dependency mechanism similar to the one used for the Self language development environment [Chambers 92]. This dependency mechanism has the same structure as the Vortex compiler's dependency mechanism, except that no method lookup caches or other internal caches or filtering nodes are used. This makes the generated code directly dependent on the program's source code and structure, without intervening filtering nodes to quash invalidations resulting from immaterial source changes. The Self system is widely regarded as having a state-of-the-art dependency system for supporting its interactive development environment.

• Vortex refers to our system as described in Section 5.3.

5.4.2 Selectivity Measurements

The expected benefit of a more complicated dependency mechanism is a reduction in the amount of recompilation after changes are made to the application's source files. Table 5.3 reports both the total number and the median number per compile of lines of source and files recompiled.

Table 5.3: Amount of Recompilation over 86 Compiles

Dependency System	Total lines recompiled	Total files recompiled	Median lines recompiled	Median files recompiled
Header per File	1,432,492	5939	26,888	108
Header per Class	1,328,766	5485	21,241	91
Self-like	399,055	1409	1384	5
Vortex	189,662	645	1146	4

Using either header file-based system, approximately seven times as many total lines would have been recompiled than were recompiled under the Vortex dependency mechanism. The Self-like dependency mechanism performs substantially better than the two header file systems, but it still would recompile twice as many total lines of code as the Vortex dependency mechanism did. The gap in the median number of lines compiled is much narrower. This is a result of a Self-like dependency mechanism usually being selective for method edits, but being much less selective for other kinds of changes such as adding a method or reorganizing the inheritance graph.

5.4.3 Space Measurements

Finer-grained dependency mechanisms can be substantially more selective than a coarse-grained header file-based alternative. However, they require more space to maintain. The table below reports on the number of nodes and edges required to represent the dependencies of the fully optimized Vortex program.

As expected, the finer-grained dependency mechanisms require substantially more space to represent than the coarser, header file-based systems. When compared against the Self sys-

Table 5.4: Space Requirements of Dependency Graphs

Dependency System	Node Count	Edge Count
Header per File	370	2146
Header per Class	1501	6857
Self-like	45,556	369,988
Vortex	81,270	258,846

tem, our system consumes more space (more nodes and fewer edges, but nodes generally consume more spance than) but is substantially more selective, as shown in Table 5.4.

5.5 Summary

We performed several experiments to assess the selectivity and space costs of our dependency mechanism, relative to both a coarse-grained `make`-like dependence mechanism and to the fine-grained dependence mechanism employed by the Self compiler, which also strives for selective recompilation. The experiments showed that our dependency mechanism was much more selective than the simple file-granularity dependencies provided by `make`, albeit at a substantial increase in space over these simple file-granularity dependencies. Compared to the Self-like dependency mechanism, the use of filtering nodes in our dependency representation allowed us to invalidate only half as much code as the Self system's mechanism, while consuming slightly more space than the dependencies used by the Self mechanism.

Using this dependency mechanism as the basis for recompilation in the Vortex compiler, we have successfully married an optimizer that performs extensive interprocedural optimizations (often depending on fairly subtle aspects of the source program) with an incremental recompilation system that allows these optimizations to be performed as part of the normal development process, rather than only being applied infrequently due to compile-time concerns. The dependency mechanism that we developed to support this seems to provide a useful point in the time vs. space vs. selectivity tradeoff. Several kinds of derived summary information are used in this system, most notably a compile-time method lookup cache, and the use of filtering nodes avoids many unnecessary invalidations of optimized code. We have used this dependency mechanism during our day-to-day development of the Vortex compiler, demonstrating that its selectivity is sufficient to permit us to make use of extensive cross-module optimization during active development of a large program

Chapter 6

Inlining Trials

Optimization techniques for object-oriented languages, including class hierarchy analysis (see Chapter 3), selective specialization (see Chapter 4), and profile-guided receiver class prediction [Hölzle & Ungar 94, Grove et al. 95], are primarily concerned with converting dynamically-dispatched message sends into statically-bound calls at compile-time, so that other optimizations can be applied. An especially important optimization that these techniques strive to enable is inlining. Inlining (also known as procedure integration and unfolding) not only confers the direct benefits of eliminating the procedure call and return sequences but also facilitates optimizing the body of the called routine in the context of the call site; sometimes these indirect post-inlining benefits are more significant than the direct benefits of inlining[1]. Much of the work in this thesis is concerned with narrowing down the targets of a message send to a single routine so that the message send can be statically bound, making it amenable to inlining. Whole-program analysis also allows inlining across source modules, further increasing the opportunities for inlining. However, with these increased opportunities for inlining comes a responsibility on the compiler's part to use inlining judiciously. Because inlining can substantially increase code space and compile time, it should only be applied in cases where the increase in performance justifies its costs.

Inlining has long been applied to languages like C and Fortran, but it may be even more beneficial in the context of higher-level languages. Functional languages such as Scheme

1. As some justification for the large role indirect benefits play in the overall performance improvement of inlining, we performed an experiment whereby we disabled a single optimization (dead closure creation elimination) in a system that performed inlining. Experiments with compiling a suite of Cecil programs using Vortex showed that the addition of this single post-inlining optimization improved performance by as much as a factor of 3 over a system that performed inlining and all other optimizations but did not perform dead closure creation elimination [Chambers et al. 96].

[Rees & Clinger 86] and ML [Milner et al. 90], pure object-oriented languages such as Smalltalk [Goldberg & Robson 83] and Cecil [Chambers 93a], and reflective systems such as CLOS [Bobrow et al. 88] and SchemeXerox [Adams et al. 93] encourage programmers to write general, reusable routines and solve problems by composing existing functionality, leading to programs with very high call frequencies. Inlining can reduce the cost of these abstraction mechanisms and thereby foster better programming styles.

Inlining can be controlled in a number of different ways. In many systems, the profitability of inlining a particular routine is hard-wired into the compiler. For example, the Smalltalk-80 compiler hard-wires the definition and optimized implementation of several basic functions from its standard library [Deutsch & Schiffman 84], and the Haskell standard prelude is fixed so that compilers can implement the functions in the standard library more efficiently [Hudak et al. 92]. A drawback of the hard-wiring approach is that built-in routines usually run much faster than user-defined routines, discouraging programmers from defining and using their own abstractions. Other systems, including C++, Modula-3, SchemeXerox [Adams et al. 93], Common Lisp [Steele 90], Similix [Bondorf 91], and Schism [Consel 90], allow programmers to indicate explicitly which routines are profitable to inline. While granting programmers fine control over the compilation process, this approach requires programmers to have a fair understanding of the language's implementation issues, and this assumption is becoming less likely as implementations become more sophisticated. Explicitly annotating routines to be inlined can also be tedious if inlining must be applied heavily to get good performance. Additionally, most explicit declaration-based mechanisms do not allow programmers to specify that inlining is profitable only in certain contexts, or that inlining should only take place within specific contexts, such as at particular high-frequency call sites of a routine, or at those call sites where static information available at the call site permits substantial optimization of the inlined code.

Another approach is to have the compiler automatically decide what routines are profitable to inline. This frees the programmer from having to make inlining decisions, but obtaining good performance requires that the compiler makes these inlining decisions well. In particular, it must strike a careful balance between over-inlining, which can dramatically lengthen compile times, and under-inlining, which can lead to code that does not perform as well. This chapter investigates techniques for automatically deciding when inlining is profitable. Making good inlining decisions depends crucially on accurately assessing the costs and benefits of inlining. As will be discussed in Section 6.1, previous automatic decision makers used simple techniques for estimating costs based on an examination of the target routine's source code or unoptimized intermediate code, and consequently they failed to take into account the effect of post-inlining optimization of the target routine. Our work corrects this deficiency, leading to more accurate cost and benefit estimates and therefore better inlining decisions.

Our system assesses the costs and benefits of inlining by first experimentally inlining the target routine, in the process measuring the actual costs and benefits of that particular inline-expansion, and then amortizing the cost of the experiment (called an *inlining trial*) across future calls to that routine by storing the results of the trial in a persistent database. Because the indirect costs and benefits of inlining can depend greatly on the amount of the static information available at the call site (e.g., the static value or class of an argument), our system performs *class set group analysis* to determine the amount of available call-site-specific static information that was exploited during optimization. Each database entry is guarded with class set group information, restricting reuse of the information derived from an inlining trial to those call sites that would generate substantially the same compiled code.

To prevent confusion, it is worth mentioning here that the development of the inlining trials technique was done before the Vortex compiler project was started. Therefore, the implementation of inlining trials was done in the context of the Self-91 compiler, and the suite of benchmarks evaluated was a set of Self programs. Although this makes it impossible to evaluate what the exact benefits of inlining trials would be if they were implemented in Vortex, it seems likely e that they would have more of an effect than they did within the Self-91 compiler. The Self-91 compiler performed fairly simplistic static optimizations, performing only intraprocedural class analysis and customization, and did not make use of profile-guided class prediction, and therefore had fewer opportunities to inline than does the Vortex compiler, which employs more sophisticated static optimizations like class hierarchy analysis and exhaustive class testing, as well as profile-guided class prediction. Examining the data presented in Chapter 3 shows that this combination of optimizations leads to a much higher percentage of dynamic dispatches being resolved at compile-time than do the simpler optimizations performed by the Self-91 compiler.

The next section reviews previous techniques for making inlining decisions automatically. Section 6.2 describes inlining trials, with class set group analysis detailed in section 6.3. In Section 6.4 we present experimental measurements of our implementation. Section 6.5 describes some other related work, and section 6.6 summarizes the material.

6.1 Previous Work on Automatic Decision Making

Existing compilers typically make automatic inlining decisions using an estimate of the cost of inlining based on an examination of the routine's unoptimized source code or intermediate representation. For example, the Self-91 compiler counts the number of message sends in the candidate routine (since message sends are the only kind of source expression) and inlines the routine if this number is below some threshold [Chambers 92]. The GNU `gcc` C compiler inlines a routine only if the number of instructions in its RTL (register transfer language) representation is less than some threshold [Stallman 90].

Source-level heuristics suffer from the problem that they do not consider the effect of optimizations applied to the body of the called routine after inlining, in particular those optimizations derived from static information available at the call site. For example, a hash table lookup routine may normally be considered too big to inline profitably. But if the key to the hash table is a compile-time constant, then some of the code of the lookup (such as computing the hash of the key) could be optimized away after inlining, making the lookup routine more attractive to inline. If the hash table itself is a compile-time constant, then the entire lookup routine can be constant folded away.

Source-level heuristics can be overly sensitive to the superficial form of the target routine. For example, the Self-91 compiler's original source-level heuristics had been tuned so that important routines, including the routine implementing a `for`-loop, were inlined. Several years later, the standard library was reorganized, and the definition of the `for`-loop routine was changed in a superficial way to be easier to read. The changed version appeared more complex to the compiler, however, and the compiler silently ceased to inline `for` loops. Performance on loop-intensive code mysteriously plummeted as a result. Such experiences, as well as only modestly-successful attempts to improve the source-level heuristics, provided the motivation for us to develop inlining trials. By assessing costs and benefits of inlining on the routine *after* optimization, inlining trials are much less sensitive to superficial details of the source code and can adapt as the source code evolves.

The Impact C compiler uses profile information to help guide the inlining process [Chang et al. 92]. The profile information is used to weight arcs in the program's call graph, allowing the cost/benefit estimates to be weighted by the expected execution frequency, and leading to better inlining decisions. Our implementation of inlining trials does not incorporate profile data, instead relying on static estimates of execution frequency, but profiling information would be easy to incorporate into an inlining trial-based system.

6.2 Inlining Trials

To make better inlining decisions, the compiler needs more accurate information on the actual costs and benefits of inlining a routine in the context of a particular call site. Accurate information can be obtained by tentatively inlining the routine, optimizing the inlined routine in the context of the call site, and then examining the resulting code. If the costs outweigh the benefits, the effects of inlining on the program representation could be undone, effectively backing out of the improper inlining decision and recovering the original uninlined state (although the compile time lost cannot be recovered, of course!). Such a conditional inline expansion, used to calculate the costs and benefits of inlining including the effects of optimization, we call an *inlining trial*.

Clearly, performing an inlining trial is much more time-consuming than estimating costs and benefits based on unoptimized source code. To regain acceptable compile-time costs, we save the results of each inlining trial in an *inlining database* that persists across compiles of a program. (The Vortex compiler's program database, as depicted in Figure 2.10, would provide an ideal location for storing the inlining database). Future opportunities to inline the same routine at other call sites consult the database instead of repeating the trial, thereby amortizing the cost of the trial over all uses of the information in the database. If a routine is called from many call sites, the amortized compile time cost of the trial can be small. Furthermore, if a few routines are identified that turn out to be bad choices to inline, the savings reaped by not inlining those routines can offset the cost of all the trials. Our experience using this approach is that many routines are invoked from multiple call sites and, as reported in Section 6.4, overall compilation time for an application actually *decreases* when using inlining trials.

The process involved in making an inlining decision using an inlining database is summarized in Figure 6.1. The remainder of this section discusses inlining trials in more detail. Sections 6.2.1 and 6.2.2 describe how we estimate costs and benefits of inlining during a trial, respectively, and section 6.2.3 discusses how to make the final inlining decision given cost and benefit information. Section 6.2.4 addresses what happens when inlining is invoked recursively within a trial. Section 6.3 explains class set group analysis, the mechanism whereby

```
$F$: the estimated execution frequency of the call site
$R$: the target routine
$T$: static information available at the call site
$TG$: class set group information describing call site-specific static
      information exploited during inlining trial
$c$, $b$: cost and benefit information for inlining trial
$D$: inlining database = $R \times TG \rightarrow c \times b$

should-inline($R$, $T$, $F$, $D$) =
  if $\exists$ ($R$, $TG$) $\in$ domain($D$) such that $T \in TG$ then
      -- use database entry if available
      ($c$, $b$) $\leftarrow$ $D$($R$, $TG$)
  else if source-level-length($R$) $\leq$ threshold then
      -- do inlining trial if simple source-level heuristic passes
      ($c$, $b$, $TG$) $\leftarrow$ perform-trial($R$, $T$)
      add (($R$, $TG$) $\rightarrow$ ($c$, $b$)) to $D$
  else
      -- don't bother with trial
      ($c$, $b$) $\leftarrow$ ($\infty$, 0)
  end
  return make-decision($c$, $b$, $F$)
```

Figure 6.1: Making an inlining decision using an inlining database

our system describes the amount of call-site-specific class information exploited when optimizing the inlined routine.

6.2.1 Estimating Costs

The major costs of inlining are increased compiled code size and increased compile times. Computing the space cost of inlining a routine is easy: after optimizing the routine in the context of the call-site, the compiler sums the expected space needed to generate machine code for each control flow graph node in the body of the inlined routine; in our implementation this is an estimate since register allocation and instruction scheduling have not yet been performed. The compiled code space needed to generate a call is then subtracted from the space taken by the inlined routine to determine the total expected space cost for inlining. This compiled code space required can be negative, if the body of the inlined routine after optimization takes up less space than the original procedure call sequence, which happens quite often in Self programs with wrapper routines that simply call other routines, and with accessor routines that just access an instance variable (and so compile down to a load instruction after inlining and optimization).

Estimating the compile time required to inline a routine is more difficult. Simply using a timer to measure compilation time suffers from the low resolution of the timing primitives exported to the user-level on many systems. It also is difficult to calibrate across different compilation platforms and across versions of the compiler with differing levels of debugging instrumentation. Fortunately, compilation time seems to be roughly proportional to compiled code space usage: we measured the compilation of 1,972 Self procedures in the Self-91 compiler and found a correlation coefficient $r = 0.93$. Consequently, our implementation considers only compiled code space usage in the cost/benefit tradeoff.

6.2.2 Estimating Benefits

The major benefit of inlining we consider is reduced execution time through elimination of executed instructions. Time savings can be viewed either as an absolute savings, such as number of dynamic instructions eliminated, or as a relative savings, such as proportion of dynamic instructions eliminated (versus a non-inlined call of the routine). Our implementation supports both views by computing two execution time estimates: *instrRemaining*, the number of instructions taken by an execution of the inlined routine and *instrSaved*, the absolute number of instructions saved as a result of inlining, after optimizations have been applied. The relative estimated execution time savings can be calculated from these two numbers as: 1 - [*instrRemaining* / (*instrRemaining* + *instrSaved*)].

Computing an estimate of time taken in an invocation of the inlined routine requires estimating the time taken for each control flow graph node, after optimization, weighting it by its expected execution frequency, and summing. This calculation is mostly straightforward, using standard compiler static estimates for execution frequency such as those described by Wagner et. al [Wagner et al. 94].

To determine the execution time saved as a result of inlining, the compiler monitors each optimization performed on the body of the inlined routine and estimates the number of dynamic machine instructions skipped as a result of the optimization, weighted by expected execution frequency. To ensure that only the benefits due to inlining are captured, the compiler considers only those optimizations enabled by static information that was available at the call site; other optimizations would be performed whether or not the routine was inlined, and so their effect should not be considered as a benefit of inlining. During an inlining trial, the compiler maintains a data structure describing the subset of available static information derived from the call site. Only optimizations based on information in the subset affect the execution time saved as a result of inlining. The savings attributed to these optimizations, plus the direct savings of the eliminated call and return sequence, form the estimated savings in execution time due to inlining. This estimate only measures the benefits of optimization on optimization of the inlined code (i.e., the benefits of optimizing the callee), and does not measure the benefits of optimizing the caller in the context of the (now-inlined) callee due to more specific information about the callee.

6.2.3 Making Final Inlining Decisions

Once the cost and benefit information for a call site has been obtained, either by performing an inlining trial or by locating an applicable entry in the database, the compiler must make a decision. This decision depends on the relative value of compile time, compiled code space, and execution time. Inlining trials provide better information upon which to base an inlining decision, but some controlling mechanism still needs to make a decision. For our implementation, we use a simple function that considers compiled code space cost and relative execution time savings and inlines the routine if the ratio of time savings to space cost is above a particular threshold; dynamic profile data could be included easily by weighting the expected execution time savings.

6.2.4 Nested Inlining

When optimizing an inlined routine, calls within the inlined routine may themselves be candidates for inlining. Optimizing these candidates can lead to recursive inlining trials. Such recursion poses no problems, and in fact occurs often in our implementation. The costs of inlining a routine include the costs associated with inlining any of its callees, and the benefits of inlining a routine include any benefits derived as part of inlining calls within the routine. The compiler must track the flow of static information from the outer call site through any contained calls, in order to correctly attribute the savings derived from some optimization to the appropriate source of static information. A slight complication is that any updates of the information for an outer inlining trial must be treated as an atomic transaction that is only committed after it is known that the costs and benefits of inlining the inner routine are such that it should have been inlined. If a nested inlining trial reveals that the inner routine should not have been inlined, then the state of the enclosing inlining trials must reflect the costs and benefits of leaving the call to the inner routine as a call. Further details of the way in which nested inlining trials were implemented to preserve this behavior within the Self-91 compiler can be found in a separate technical report [Dean & Chambers 93].

6.3 Class Set Group Analysis

During an inlining trial, the compiler uses any information available statically at the call site to optimize the body of the inlined routine. Consequently, the costs and benefits of the trial reflect this call-site-specific information. For example, if at some call site the compiler has information about the possible class(es) of an argument, message sends to the argument in the body of the inlined routine may be optimized substantially, increasing the benefits of inlining the routine. However, a different call site that lacked static information about the argument's class would be attributed a lower inlining benefit. If the results of an inlining trial for one of these two call sites were applied to the other, inappropriate inlining decisions might be made.

To avoid these potential problems, an inlining trial database entry is guarded with a description of the kind of static information that should be present at the candidate call site for the results of the trial to be reasonably predictive. During an inlining trial, the compiler monitors uses of static information derived from the caller and records the amount of static information that enabled (or disabled, in the case of a lack of static information) each optimization. This summary information is added to the inlining database entry storing the results of the trial. When a future call site searches the inlining database, the static information available at the call site must be compatible with the summary of an entry for it to match.

In the Self-91 system, static class information about the arguments to the inlined call is the principal source of information that drives optimizations. Guarding an inlining trial's database entry with the actual static class information available about the arguments would be too specific, however: few call sites would have exactly the same static class information as the inlining trial, and consequently there would be little reuse of the results of inlining trials. Instead, the inlining entry should be guarded with a *description* of the kinds of static information that lead to roughly the same degree of optimization. We call these descriptions of static information *class set groups*. A class set group specifies a set of class sets, where all member class sets of classes lead to substantially the same optimizations being performed as part of inlining. As in Chapter 3, class sets themselves describe sets of values that share common properties relevant to the optimizations performed by the compiler. For the purposes of inlining trials, information about values being particular constants is tracked as a special form of class information. In the same way that class sets represent sets of values, class set groups represent sets of class sets. Table 6.1 describes the class set groups used in our implementation of inlining trials (T stands for the set of all class sets):

Table 6.1: Kinds of class set group information

Class Set Group name	**Set description**	**Meaning**
Universal	T	Any class information
SubsetGroup($s \in T$)	$\{ t \in T \mid t \subseteq s \}$	Any class set which is at least as precise as s
AClass	$\{ t \in T \mid t \subseteq C, C$ a class set $\}$	Any class set with class-level information
AClosure	$\{ t \in T \mid t$ is a closure $\}$	Any closure (a special kind of class information)
AConstant	$\{ t \in T \mid \lvert t \rvert = 1 \}$	Any class describing a compile-time constant
IntersectGroup($t_1, ..., t_n$)	$t_1 \cap ... \cap t_n$	Intersection of several class set groups
ExcludeGroup($s \in T$)	$\{ t \in T \mid t \not\subset s \}$	Any class not in a certain class set group

6.3.1 Using Class Set Group Information

Each argument of a database entry is guarded with a class set group. For a database entry to be applicable to the call site, the static class information for each actual argument must be a member of the set specified by the corresponding class set group. If for example the class set group of some argument is the Universal class set group, then any actual argument class will match; this implies that the optimization of the inlined routine does not depend on the static information available for that argument. If instead the class set group was IntersectGroup[SubsetGroup(Class(Integer), AConstant)], then only actual arguments whose static information conveyed that the argument was some integer constant would match. Such a precise class set group implies that the compilation of the inlined routine is able to exploit the information that the argument is some integer constant, say through constant folding within the inlined routine, that would not be possible if less static information were available. As a final example, if the class set group were ExcludeGroup(AClass), then only class sets that did not have information that was as precise as knowing the exact class would match. Class set groups that exclude more precise kinds of static information ensure that inlining candidates do not match against database entries for trials that were unable to perform optimizations due to a lack of static information at the call site. In this specific example, the lack of specific class-level information during the trial prevented some optimization, such as performing message lookup at compile-time or eliminating a run-time class check.

6.3.2 Computing Class Set Group Information

To compute class set group information for each argument, the compiler performs *class set group analysis*. Class set group analysis is unusual in that it does not compute some abstraction of the values manipulated by the program being compiled, but rather it monitors the compilation process itself, computing how the compiler manipulates static class information. From this standpoint, class set group analysis is a kind of *meta-analysis*.

Class set group analysis is performed in parallel with regular class analysis during an inlining trial. At the beginning of a trial, each argument to the inlined routine is associated with the Universal class set group, indicating that, so far, no static information about the arguments has been used. Whenever an optimization is performed based on static class information derived from an argument, the class set group associated with that argument is narrowed by intersecting it with a class set group that represents the kind of static information that enabled the optimization. Similarly, whenever an optimization is *disabled* because of a lack of precision in the static class information known about an argument, the class set group for that argument is intersected with an ExcludeGroup class set group that rules out class sets that could have enabled the optimization. Table 6.2 indicates, for some of the more common optimizations performed in the Self-91 compiler, the class set group intersected if the static information about the argument enabled or disabled the optimization.

The class set groups calculated as part of class set group analysis are intended to represent the largest set of argument types that would lead to the same optimizations being performed at a future call site.

Table 6.2: Effects of optimizations on class set group information

optimization	if enabled	if disabled
perform message lookup at compile-time to statically-bind a message	SubsetGroup({ *least specific info allowing static binding* })	ExcludeGroup(SubsetGroup({ *least specific info allowing static binding* })
constant folding	AConstant	ExcludeGroup(AConstant)
eliminate integer, float, etc. class tests	SubsetGroup(Class(*the class*))	ExcludeGroup(AClass)
eliminate true, false value tests, constant fold operation	AConstant	ExcludeGroup(AConstant)
inline-expand body of closure	AClosure	ExcludeGroup(AClosure)

```
method fetch(self@growable_sequence, index) {
   if index < 0 or index > self.max_index then
      error("index out of bounds")
   endif
   return self.elems[index + self.base_index]
}
... seq.fetch(i) ...
```

Figure 6.2: Example inlining candidate

6.3.3 An Example

We will use the inlining candidate shown in Figure 6.2 to illustrate how class analysis and class set group analysis interact. The example assumes that the compiler knows the exact class of the `seq` variable statically and it has statically-bound the `seq.fetch` message to the fetch implementation shown in Figure 6.2. We will first consider the case where the compiler has static information that shows that argument `i` is of class integer. The compiler consults the inlining database for a matching entry. For the purposes of the example, we will assume that such an entry does not exist. Since the target method is not unreasonably large, the compiler begins an inlining trial. Initially the class set group associated with the argument `index` is Universal; no optimizations yet exploit any static information about `index`. The first operation within the routine sends the < message to `index`. The compiler examines the static class information known about `index`, discovers that the class of `index` is known statically, and statically-binds and inline-expands the `integer::<` method (perhaps invoking a recursive inlining trial in the process). To reflect using class-specific information about the `index` argument, the compiler narrows the class set group of `index` from Universal to IntersectGroup(Universal, SubsetGroup(Class(integer))), or simply SubsetGroup(Class(integer). The compiler also updates the benefit information for the trial to reflect saving more than a dozen instructions by eliminating the overhead of dynamic binding and the call/return sequences for the < message.

The compiler then analyzes the body of the inlined < method. The < method for integers first tests that its argument's class is also an integer. It is, but since the second argument to < is not being monitored as part of the inlining trial for `fetch`, no class set group information is affected (for simplicity, we are assuming that we are not performing a nested inlining trial during the inlining of the < method; had we been doing so, then we would have updated the class group information for the second argument for this nested trial). After verifying that its arguments are integers, the compiler attempts to constant-fold the comparison. This requires both arguments to be integer constants, which does not succeed. The compiler again narrows the class set group information for `index` to indicate that its static information was not specific enough to enable the optimization, by intersecting `index`'s class set group with ExcludeGroup(AConstant) to give IntersectGroup(SubsetGroup(integer), ExcludeGroup(AConstant)). This class set group matches all class sets that have static information that index is of class integer, but that do not have static information that index is an integer constant. Such class sets include Class(integer) and Union(Constant(3), Constant(4)) but excludes Constant(17), Unknown, and Union(Class(integer), Class(float)). Note that the class set group of `index` excludes class information that `index` is a constant, but clearly it does not exclude integer values reaching that part of the program. Class set group information can exclude overly specific *class set* information, but the *values* described by the excluded class sets can still appear, as long as some more general class set including the value is included in the class set group.

The compiler visits each of the remaining operations in the inlined routine, but no additional narrowing of the class set group of `index` occurs; additional time savings accrue, however, during optimization of the > and + messages. The compiler then completes the trial by creating a new database entry that records the compiled code size of the inlined `fetch` method, the expected cycle count of an execution of the inlined method, and the expected number of cycles saved as a result of inlining the fetch method. This entry is added to the database, guarded by the class set group calculated for the `index` argument. Finally, the compiler makes a decision about whether the fetch method should be inlined, undoing the effects of the trial if not.

Subsequent statically-bound invocations of this `fetch` method examine this database entry. If the class information known about the `index` argument at these call sites is sufficient to show that index's class is integer but index is not known to be an integer constant, then the results of the database entry are consulted to determine whether inlining is warranted. If `index` is known statically to be a particular integer constant, then a new inlining trial is performed. During such a trial, the `index < 0` expression can be constant-folded, resulting in additional savings in execution time and compiled code space that might change the decision about whether the call site is profitable to inline. Similarly, if the static class information known about the argument is less specific than Class(integer), or is some other unrelated class information, then a new inlining trial is performed to assess the costs and benefits of inlining when different static class information is known about index. An optimization to avoid performing additional inlining trials for cases that have more specific static information would be to perform an additional comparison of the static information with the database entry's class set group information omitting the ExcludeGroup components: if the database entry indicated that inlining was warranted, then the additional static information will only serve to enable additional optimizations. Similarly, if a database entry indicates that inlining is not desirable with a given level of static information, then call sites with even less static information will not be desirable to inline.

Without some mechanism like inlining trials and class set groups, the compiler could examine only the unoptimized source code for the `fetch` method. In this and many similar cases, the kind of static information about the arguments to the call can have a significant effect on the nature of the final code; some calls will be profitable to inline, while others will not be. Inlining trials provide the compiler with more accurate information upon which to make decisions, and class set groups enable the compiler to distinguish among call sites with different available static information.

6.4 Experimental Results

We implemented and measured this approach in the context of the Self-91 optimizing compiler [Chambers & Ungar 91]. Self [Ungar & Smith 87] is a pure object-oriented language similar to Smalltalk but without any hard-wired operations or control structures, and shares many characteristics with Cecil. The Self-91 compiler exploits dynamic compilation, interleaving compilation with execution, to get fast turnaround times and to benefit from a form of profile information. As a result, keeping compile times short is of particular importance. However, because of Self's pure language model, the compiler must inline aggressively to obtain reasonable performance. By replacing the original heuristics for making automatic inlining decisions with an inlining trial-based approach, we sought to reduce compilation time while retaining the same level of run-time performance. Inlining trials were effective at this task: for six Self programs, compile time was reduced by an average of 20% with virtually no loss in run-time performance. We believe that in systems with more opportunities to inline than the Self-91 compiler we studied, inlining trials and class set group analysis could make an even bigger improvement in the compile-time/run-time tradeoff.

As a first step of assessing the effectiveness of inlining trials, we compared our new inlining decision making system using inlining trials against the source-level heuristics found in the existing Self-91 compiler, measuring compilation time, execution time, and compiled code space consumption. Since we suspected that the original source-level heuristics were overinlining, we set the initial "reasonably short" threshold that identifies routines where performing an inlining trial seems feasible to exactly the same value used by the source-level heuristics. We were interested in seeing if we could obtain the same level of runtime performance with less compile time cost, by avoiding the inlining of poor inlining candidates but still inlining the good inlining candidates. Thus the only difference in the first set of experiments is is that the new inlining-trial-based system might choose not to inline something that the existing system would inline. We examined the behavior of inlining trials on a suite of Self programs ranging in size from 300 to 12,500 lines.[2] The programs are described in Table 6.3.

We suspected that the existing heuristics, tuned initially on small benchmarks, over-inlined for these larger programs, which would lead to slower compiles and more space-consuming compiled code without much benefit in execution speed. We hoped that inlining trials would make better decisions on what routines were profitable to inline. Figure 6.3 reports the compilation time, execution time, and code space usage of these six programs for our new system, relative to the existing system.[3] Shorter bars indicate better performance for the new system.

2. We also examined a large number of small benchmarks, used during the original development of the Self-91 compiler. The existing heuristics had been tuned to make good decisions for these small benchmarks, and the inlining trial-based system achieved the same compile-time and run-time performance as the existing system, as we hoped.

3. The values in the chart are calculated as the compilation time, execution time, and compiled code space usage for the new system divided by that for the existing system, converted to a percentage. Execution time denotes just the time spent executing compiled code, not the time spent compiling the code.

Table 6.3: Benchmark programs for inlining trials

Program	Size (lines)[a]	Description
parser	400	Parser for an old version of Self
primMaker	1,300	Program to generate wrapper functions from an interface description file
pathCache	300	Traverses the Self object graph and assigns path names to objects
deltaBlue	600	Incremental constraint solving program
cecilInterp	10,700	Interpreter for an old version of the Cecil language
cecilCompiler	12,500	Compiler for an old version of the Cecil language

a. Excluding standard library of 30,000 lines

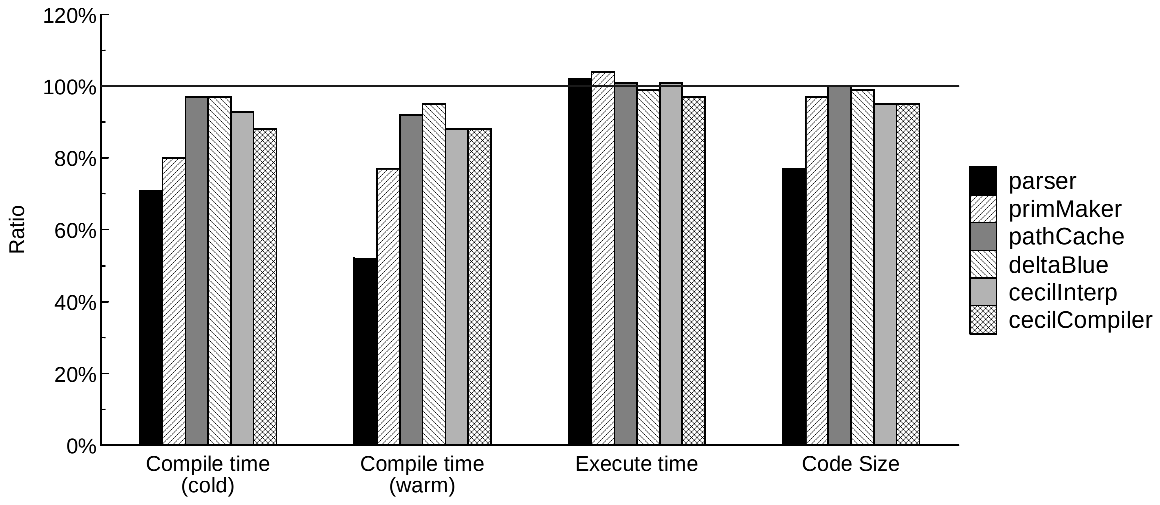


Figure 6.3: Ratio of inlining trials to old heuristics

The chart shows compilation times both for starting with an empty inlining database for each program (“cold”) and for starting with a filled inlining database (“warm”). The warm compiles were measured by reusing the database generated during the “cold compile” for the benchmark. In practice, since the database is persistent and entries are shared across programs, the compilation performance is closer to the warm compile figures than to the cold compile figures.

On average (using geometric mean), compilation time decreased by 20%, execution time increased by 1%, and code space usage decreased by 6%. On an absolute scale, the compilation time savings of 20% represent a savings of 68 seconds of the 291 seconds required to compile all six programs; in our environment, compilation time is a significant cost worthy of optimization effort. Based on these results, we consider inlining trials to be effective at meeting our goal of balanced compilation and execution times.

The parser program shows particularly good improvement in compilation time. Under the old source-level heuristics, the `advance` routine, called to move the current character position forward in the input buffer, was inlined 26 separate times. However, `advance` does not benefit much from static class information available at the call site, so there is little indirect benefit from inlining. The inlining trial-based system detected this and consequently never inlined the `advance` routine, saving a substantial amount of compilation time and code space in the process.

The compilation improvement shown by these programs, while quite significant, is not as impressive as it might be in another environment. For these programs, the current Self-91 compiler is unable to statically bind many messages because of a lack of static class information [Chambers 92]. More recent compilers, such as the Vortex compiler and the Self-93 compiler, incorporate more sophisticated analyses and transformations to statically-bind message sends, leading to many more messages being statically bound and thus eligible for inlining. In such compilers, the importance of making good inlining decisions increases, as the number of opportunities to inline increases.

The above experiments used the same initial threshold for both systems. As a second set of experiments to see how sensitive the two approaches are to the choice of this threshold, we repeated the comparison of the two systems for a range of thresholds. Figure 6.4 shows the geometric mean of compilation time and execution time for the two systems on the six-program benchmark suite for several different thresholds. The existing source-level inlining heuristic is computed by summing weighted values for non-trivial message sends within the target routine. Certain messages which the compiler expects to be optimized (such as “+” and “`at:`”) are assigned a weight of 1 and other message sends are assigned a weight of 2.[4] A routine is eligible for inlining if the weighted sum of its messages is less than or equal to the inlining threshold. The values in the chart have been normalized to the performance of the old inlining heuristics when using the default threshold of 8. Increasing the threshold value increases the number of routines considered for inlining.

4. These source-level heuristics were developed after careful consideration of the inlining process and represent an improvement over an older version of the source-level heuristic that did not consider common messages specially. This heuristic represents our best effort to date at a reasonable source-level inlining heuristic.

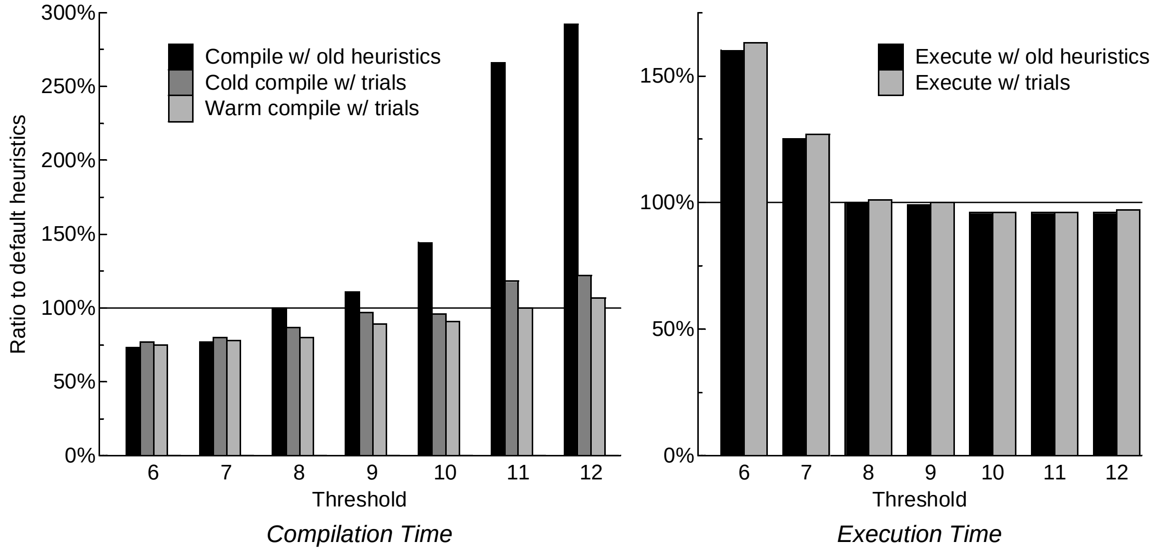


Figure 6.4: Effect of varying inlining threshold

Compilation time is much less sensitive to the choice of threshold under the new inlining trial-based heuristics than under the old source-length heuristics, and the new approach has significantly better compilation time behavior than the old system. Also, the new inlining trial-based decision-making achieves nearly the same execution speed as under the old heuristics. Together, these results illustrate some of the different compile-time/run-time tradeoffs that can be made. In our system we set the threshold to 8, leading to a 20% reduction in compilation time with a negligible loss of execution speed. If instead we set the threshold to 10, compilation time would still drop by 9% but run time would also drop by 4%. Because compilation speed does not degrade much when using a higher initial threshold under the new system, we can use a higher threshold and be more robust in the face of future superficial changes to the source code of libraries and applications, such as the superficial rewrite of the `for`-loop implementation described in Section 6.1.

Figure 6.5 reports the number of database entries generated by compiling the programs. The first six columns represent the number of entries created when compiling each program individually against an initially empty database. In our implementation, each database entry takes up approximately 75 bytes of space; the savings in compiled code space for using inlining trials compensates for the additional space cost of the database, and the compiled code space savings persist after program development ceases. The rightmost column indicates the total number of entries generated by compiling the six programs in succession, starting with

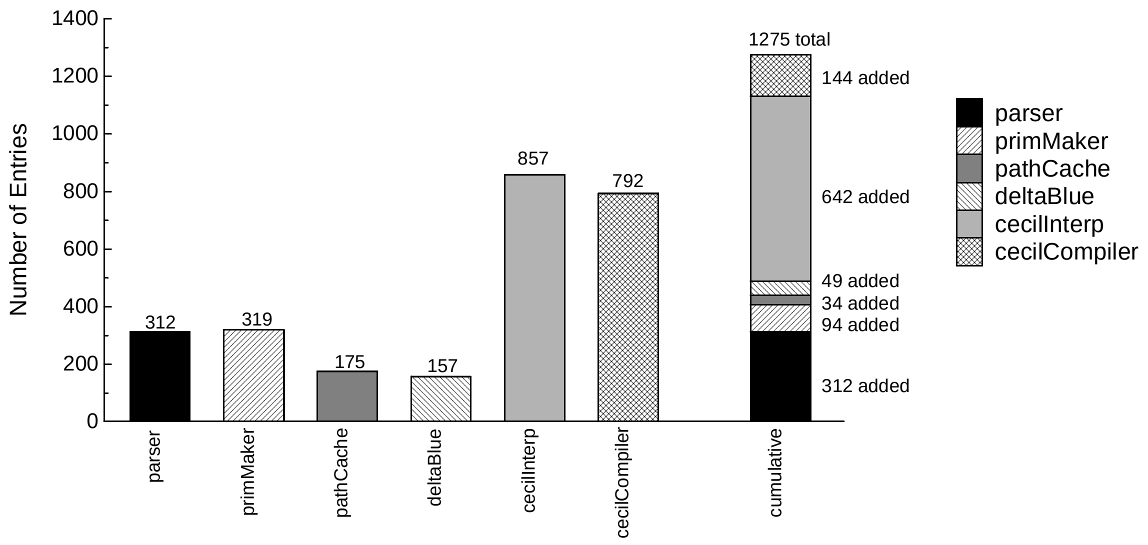


Figure 6.5: Inlining trial database sizes

an initially empty database. The numbers to the right of this column indicate the number of new entries generated by each program in this successive compilation. Because many database entries representing entries for functions in the shared standard library are used by more than one of the programs, the total number of entries generated by compiling all six programs in succession (1275) is only half of the sum of the number of entries generated by compiling each program separately (2612). Inlining trials were performed on less than 2% of the inlining decisions in either case. Part of this infrequency may be attributed to Self's use of user-defined routines for basic control structures, which meant that the database entries for routines like `if:True:` and `while:` were reused many times.

6.5 Related Work on Inlining

Previous work on automatic inlining has focused primarily on attempting to maximize the direct benefits of inlining without too much increase in compiled code space [Scheifler 77, Allen & Johnson 88, Chang et al. 92]. In the context of this related work, indirect benefits of inlining tend to be relatively unimportant. Automatic inliners for higher-level functional and object-oriented languages have quite a different flavor, particularly because many things that would be built-in operators and control structures in lower-level languages tend to be user-defined in higher-level languages, and these user-defined routines need to be inlined aggressively to get good performance. Additionally, in the context of higher-level languages, the indirect benefits of inlining often are more important in determining profitability than the simple direct costs.

Ruf and Weise describe a technique for avoiding redundant specialization in a partial evaluator for Scheme [Ruf & Weise 91]. When specializing a called routine using the static information available at a call site, their technique computes a generalization of the actual types that still leads to the same specialized version of the called routine. Other call sites with different static information can then share the specialized version of the called routine, as long as they satisfy the same generalization. Our class set group analysis computes similar summary information about argument types.

Cooper, Hall, and Kennedy present a technique for identifying when creating multiple, specialized copies of a procedure can enable optimizations [Cooper et al. 92]. They apply this algorithm to the interprocedural constant propagation problem. To reduce the number of specialized copies of a procedure, their system evaluates when merging two specialized versions of a procedure would not sacrifice an important optimization. Our class set group guards on database entries accomplish a similar task, enabling the results of an inlining trial to be reused for those call sites where similar optimizations are enabled, but over a richer domain of types.

6.6 Summary

Inlining trials are a promising mechanism for gathering more accurate information about the costs and benefits of inlining in an optimizing compiler. Better information can in turn lead to better automatic decisions about which call sites to inline. If these automatic decisions are good enough, standard library routines won't need to be hard-wired into the compiler for performance and programmers won't need to annotate routines with explicit inline directives. Ultimately, good automatic inlining can foster a better programming style by making the use of abstraction less expensive.

Unlike standard source-level inlining heuristics, inlining trials can consider the effect of post-inlining optimizations when assessing the costs and benefits of inlining. This provides the compiler with more accurate data upon which to base its inlining decision, and the post-optimization data is much less sensitive to superficial details of the source code. By storing the results of trials in a persistent database, the extra cost of a trial can be amortized across uses of the information. Class set group analysis is essential for ensuring that database entries are reused for exactly those call sites whose static information would lead to the same set of optimizations being performed. Class set group analysis may be applicable to other compilation problems, such as deciding when procedure specialization is profitable.

We have applied the ideas of inlining trials and class set group analysis to improving the response time of the optimizing Self-91 compiler. In our implementation, the use of inlining trials cut compile time by 20% with virtually no effect on execution speed. By changing the cost/benefit tradeoff embodied by the final inlining decision-maker, we could have saved both compile time *and* execution time by making more intelligent inlining decisions. The extra compile-time cost of inlining trials is more than paid for by avoiding over-inlining. Incorporating dynamic profile information could improve the results even more.

Inlining trials and class set group analysis appear most useful for languages where procedural abstraction is used heavily, where the compiler can often determine statically the single target of a call, and where the effects of post-inlining optimizations are substantial and can vary across call sites. Many high-level functional and object-oriented languages meet this description. As the analyses of the targets of call sites improve, the compiler will have more opportunities to inline and consequently will bear more responsibility for making wise decisions. Inlining trials are a useful means of ensuring that inlining is only applied in the contexts in which it is most important.

Chapter 7

Conclusions

Whole-program analysis is an enabling technique for a wide variety of compiler optimizations. The "closed world" assumption that it allows is especially important for object-oriented languages where the possibility of extension prevents many kinds of optimization in the "open world" of separate compilation. Many new optimizations are enabled by this knowledge, but not all of them are equally useful. Devising practical techniques that can be applied to large programs, that have acceptable compile-time and code-space requirements, and that can be applied incrementally (such as in an interactive programming environment), requires a careful balancing of the optimization cost/benefit tradeoff. This thesis presented *class hierarchy analysis*, *exhaustive class testing*, and *selective specialization*. These three techniques provide substantial performance improvements across a wide variety of object-oriented programs and languages, and do so in such a way that they can be applied for everyday use. The development of the Vortex compiler serves both as a reference implementation of these techniques and as a demonstration of their practicality, since Vortex itself was used as the compiler for all of its development. Supporting work presented in this thesis includes techniques for maintaining dependencies to support selective recompilation and the new technique of *inlining trials*. Both of these techniques serve to make whole-program optimizations more practical by reducing the compile-time required to reach effective levels of optimization.

Compared with a separate-compilation model, whole-program analysis and its closed-world assumption provides a great deal more optimization opportunities, especially for language features like message dispatching.

7.1 Future Directions

Further improving the performance of object-oriented languages is an important and active area of research. The increasing popularity of more-flexible object-oriented languages such as Java and Smalltalk and the use of hybrid languages like C++ and Modula-3 in a more object-oriented manner means that optimization techniques that provide good performance for programs written in a flexible, extensible style are becoming increasingly important. With the ever-increasing computing power available on machines used for software development, techniques such as whole-program optimization, unthinkable for large programs a decade ago, become practical. This thesis has explored some of the opportunities and issues related to whole-program optimization, but there are still unexplored areas, including more powerful interprocedural analyses and transformations, and the exploration of language design in the presence of whole-program optimization.

7.1.1 More Sophisticated Interprocedural Optimizations

Much of the performance problems of object-oriented languages stem from the preponderance of message sends, and, as demonstrated by the experiments with “perfect” class information in Section 3.4.3, interprocedural class analyses have the potential to eliminate much of the remaining overhead from programs. Class hierarchy analysis is perhaps the simplest and most scalable interprocedural class analysis algorithm that one can imagine, and it has been shown to be remarkably effective for programs that make heavy use of dynamic dispatching. More sophisticated algorithms that perform flow-insensitive and flow-sensitive analyses of the entire program’s source code (rather than just examining the hierarchy of class and method declarations) promise to provide more-precise information and even better performance but currently have problems, both in terms of analysis time and in space required by the analysis, when applied to large programs [Grove 95]. Existing interprocedural class analysis algorithms are also not incremental, meaning they recompute all their results every time the program source changes. Devising interprocedural class analysis algorithms that both scale well when applied to large programs and that can be applied incrementally is an important step, not just because of the more precise class analysis results provided by the algorithms, but also because these algorithms compute an accurate call graph as a by-product. A call graph is a required starting point for many other kinds of interprocedural optimizations and transformations. One such optimization is the automatic specialization of an object’s layout by unboxing primitive datatypes and eliminating pointer indirections to component objects. Similar optimizations have been shown to greatly improve the performance of ML programs [Shao & Appel 95, Tarditi et al. 96], but applying this optimization to object-oriented languages requires a call graph to track the flow of data structures through the program, in lieu of a static type system like ML's. Other more traditional analyses such as pointer analysis [Chase et al. 90, Landi & Ryder 92, Hendren et al. 92, Deutsch 94, Wilson & Lam 95] and escape analysis (object lifetime analysis) [Hudak 86, Kranz et al. 86] also rely on the program call graph.

7.1.2 Hybrids with Separate Compilation

Whole-program analysis is merely one endpoint of a spectrum from full whole-program analysis, where any form of intermodule optimization can take place, all the way to true separate compilation, where no intermodule optimization takes place. One major objection to a pure whole-program optimization approach is that it precludes the use of separately-compiled, sharable libraries. To overcome this limitation, it would be useful to explore different points along this spectrum, where a library could be compiled and optimized against a description of how an application is allowed to extend the library, and vice versa for applications against partial implementation descriptions of libraries. When merging an application and a library, if the application fulfills the assumptions made when compiling the library, and vice versa, then linking succeeds, otherwise some fix-up code would be generated to replace code compiled with violated assumptions. In this way, separately-compiled sharable libraries could be produced without sacrificing most optimization of the libraries, at some cost in lost optimization opportunities for some clients and extra fix-up code space for other clients.

7.1.3 Synergy with Language Design Research

The poor performance achieved when compiling expressive languages like Cecil using traditional compilation techniques served as a large part of the motivation for this thesis and for the use of whole-program optimization. In effect, Cecil's aggressive language design presented a number of challenges for developing an efficient implementation, driving the language implementation research forward. This motivational force can also be applied in the opposite direction. Using the whole-program optimization tools developed in this thesis, we can turn our attention to language design to explore what advances can be made in designing more-flexible and more-uniform languages that still perform well. For example, one of the design principles adhered to for C++ was that programmers should “pay only for what they use” [Stroustrup 91]. Unfortunately, because the “pay only for what you use” tradeoffs were made assuming less-aggressive implementation strategies, many flexible and useful language features were omitted. By relying on whole-program optimization, many of these features could be added to C++ (or Java or Modula-3) in such a way that these new features incur no performance cost when their added flexibility is not used. Examples of such features include:

- *Uniform treatment of all datatypes as classes.* Hybrid languages distinguish between built-in, non object-oriented datatypes and associated operators and user-defined object-oriented classes and methods, primarily for efficiency reasons. With class hierarchy analysis, all datatypes can be classes and all operations can be implemented as dynamically-dispatched messages, resulting in a more uniform languages. This would allow built-in datatypes to coexist with other kinds of objects in polymorphic data structures. Such an approach would incur costs only when the added flexibility of being able to mix datatypes resulted in sending messages that class hierarchy analysis could not resolve to a single method.

- *No need for virtual vs. non-virtual methods.* All methods can be virtual with no performance cost, since class hierarchy analysis identifies those call sites where the flexibility of dynamic dispatching is unnecessary and eliminates the cost of this flexibility. When the dynamic dispatching becomes necessary because a new subclass overrides the existing version of the operation, the compiler transparently takes care of automatically “reinstating” the flexibility, freeing the programmer to concentrate on other tasks.

- *Multi-methods.* Multi-methods unify the notions of procedures, methods, and overloaded functions into a single notion of sending a message, but the potential for dispatching on multiple arguments imposes significant costs. With whole-program analysis, however, the compiler knows exactly what messages require more sophisticated multi-method dispatching schemes, and can use standard single-dispatching techniques, such as lookup tables, for messages that are only singly-dispatched, meaning that the cost of multi-methods is only paid where their added flexibility is used.

All of these potential changes adhere to the general language design principle of allowing a great deal of flexibility at the language level, and relying on sophisticated implementation techniques to ensure that the potential for this flexibility has little or no performance cost except where it is used. The ability to detect where flexibility is unnecessary is perhaps the greatest advantage of the closed-world view provided by whole-program analysis. This thesis demonstrates the substantial advantages that can be gained from whole-program analysis, and it is hoped that future programming environments and compilers will utilize this model to improve performance, both to improve the performance of existing languages and to enable the use of more flexible programming languages whose implementations still obtain high levels of performance. The potential of whole-program analysis has just begun to be exploited.

Appendix A

Detailed Data

This appendix contains the raw data for the experiments described in Chapter 3.

Table A.1: Execution Time (seconds)

Program	native	base	inline	intra	intra+ CHA	intra+ CHA+ exh	intra+ profile	intra+ CHA+ profile	intra+ CHA+ exh+ profile
instr sched		21.11	21.11	9.69	6.58	5.32	3.11	2.61	2.74
typechecker		333.82	338.82	103.44	64.79	50.55	33.84	30.08	29.78
vortex		3,617	3,617	1,500	903	700	615	515	486
java-cup	5.0	0.85	0.82	0.72	0.50	0.49	0.69	0.50	0.47
javac	62.00	10.21	10.02	9.70	8.43	8.17	9.17	8.32	8.08
m2tom3	23.50	23.42	22.29	21.87	21.36		21.91	21.92	
prover	28.90	32.75	31.54	32.25	31.55		32.02	32.39	
m3fe	21.90	22.78	22.91	22.28	21.90		22.16	22.13	
ixx	0.86	0.92	0.80	0.79	0.70		0.74	0.70	
ktsim	106.17	107.41	99.28	98.47	96.84		97.17	96.52	
eon	76.22	82.60	70.54	65.39	63.78		64.12	64.41	
porky	9.75	13.56	13.00	13.13	12.72		12.86	12.67	
ixx-av	0.88	0.94	0.83	0.85	0.71		0.79	0.71	
ktsim-av	113.60	116.67	107.73	108.24	100.60		107.42	98.98	

Table A.2: Dynamic Number of Message Sends (x1000)

Program	native	base	inline	intra	intra+ CHA	intra+ CHA+ exh	intra+ profile	intra+ CHA+ profile	intra+ CHA+ exh+ profile
instr sched		9,926	9,926	5,663	2,577	1,599	466	399	377
typechecker		117,899	117,889	62,357	22,382	13,713	6,961	6,410	5,605
vortex		1,097,784	1,097,784	617,005	246,662	153,559	124,909	90,153	71,686
java-cup	310	310	310	273	96	71	213	61	41
javac	2,555	2,555	2,555	2,093	965	814	1,586	725	691
m2tom3	2,804	2,804	2,804	2,710	844		1,736	798	
prover	1,255	1,255	1,255	1,214	1,012		1,110	1,011	
m3fe	564	564	564	564	564		427	427	
ixx	96	96	96	96	56		4	4	
ktsim	716	716	716	716	502		342	159	
eon	7,401	7,401	7,401	7,401	6,861		3,402	3,402	
porky	3,700	3,700	3,700	3,697	3,368		1,419	1,345	
ixx-av	387	387	387	386	56		6	4	
ktsim-av	45,803	45,803	45,803	45,803	502		44,168	159	

Bibliography

[Adams et al. 93] Norman Adams, Pavel Curtis, and Mike Spreitzer. First-Class Data-Type Representations in SchemeXerox. *SIGPLAN Notices*, 28(6):139–146, June 1993. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[Agesen & Hölzle 95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages. In *OOPSLA'95 Conference Proceedings*, pages 91–107, Austin, TX, October 1995.

[Agesen & Hölzle 96] Ole Agesen and Urs Hölzle. Dynamic vs. Static Optimization Techniques for Object-Oriented Languages. *Theory and Practice of Object Systems*, 1(3), 1996.

[Agesen 95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Agesen et al. 93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzback. Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings ECOOP '93*, July 1993.

[Agrawal et al. 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In *Proceedings OOPSLA '91*, pages 113–128, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Aigner & Hölzle 96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *Proceedings ECOOP '96*, Linz, Austria, August 1996. Springer-Verlag.

[AK et al. 89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[Allen & Johnson 88] Randy Allen and Steve Johnson. Compiling C for Vectorization, Parallelization, and Inline Expansion. *SIGPLAN Notices*, 23(7):241–249, July 1988. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[Amiel et al. 94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *Proceedings OOPSLA '94*, pages 244–258, Portland, OR, October 1994.

[AT et al. 96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Rober Wahbe. Efficient and Language-Independent Mobile Programs. *SIGPLAN Notices*, pages 127–136, May 1996. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[Auslander et al. 96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan Eggers, and Brian Bershad. Fast, Effective Dynamic Compilation. *SIGPLAN Notices*, pages 149–159, May 1996. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[Bacon & Sweeney 96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Bieman & Zhao 95] James M. Bieman and Josephine Xia Zhao. Reuse Through Inheritance: A Quantitative Study of C++ Software. In *Proceedings of the Symposium on Software Reusability*. ACM SIGSOFT, August 1995. Software Engineering Notes.

[Bobrow et al. 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, 28(Special Issue), September 1988.

[Bondorf 91] Anders Bondorf. Similix Manual, System Version 4.0. Technical report, University of Copenhagen, Copenhagen, Denmark, 1991.

[Calder & Grunwald 94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, January 1994.

[Caseau 93] Yves Caseau. Efficient Handling of Multiple Inheritance Hierarchies. In *Proceedings OOPSLA '93*, pages 271–287, October 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.

[Chambers & Ungar 89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for Self, A Dynamically-Typed Object-Oriented Programming Language. *SIGPLAN Notices*, 24(7):146–160, July 1989. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[Chambers & Ungar 90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. *SIGPLAN Notices*, 25(6):150–164, June 1990. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings OOPSLA '91*, pages 1–15, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Chambers 92] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, March 1992. Published as technical report STAN-CS-92-1420.

[Chambers 93a] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report TR-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.

[Chambers 93b] Craig Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Chambers et al. 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self – a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings OOPSLA '89*, pages 49–70, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[Chambers et al. 95] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In *17th International Conference on Software Engineering*, Seattle, WA, April 1995.

[Chambers et al. 96] Craig Chambers, Jeffrey Dean, and David Grove. Whole-Program Optimization of Object-Oriented Languages. Technical Report TR-96-06-02, Department of Computer Science and Engineering. University of Washington, June 1996.

[Chang et al. 92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided Automatic Inline Expansion for C Programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[Chase et al. 90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[Chen et al. 94] Weimin Chen, Volker Turau, and Wolfgang Klas. Efficient Dynamic Look-up Strategy for Multi-Methods. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 408–431, Bologna, Italy, July 1994. Springer-Verlag.

[Consel 90] Charles Consel. *The Schism Manual, Version 1.0*. Yale University, New Haven, Connecticut, December 1990.

[Cooper et al. 92] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure Cloning. In *Proceedings of 1992 IEEE International Conference on Computer Languages*, pages 96–105, Oakland, CA, April 1992.

[Dahl & Myhrhaug 73] O.J. Dahl and B. Myhrhaug. Simula Implementation Guide. Technical Report Publication S47, Norwegian Computing Center, Oslo, Norway, March 1973.

[Dean & Chambers 93] Jeffrey Dean and Craig Chambers. Training Compilers to Make Better Inlining Decisions. Technical Report TR-93-05-05, Department of Computer Science and Engineering. University of Washington, May 1993.

[Dean et al. 95a] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. *SIGPLAN Notices*, pages 93–102, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[Dean et al. 95b] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Dean et al. 96] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Deutsch & Schiffman 84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, January 1984.

[Deutsch 94] Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[Diwan et al. 96] Amer Diwan, Eliot Moss, and Kathryn McKinley. Simple and Effective Analysis of Statically-typed Object-Oriented Programs. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Driesen & Hölzle 96] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Driesen et al. 95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Dyl92] Dylan, an Object-Oriented Dynamic Language, April 1992. Apple Computer.

[Feldman 79] Stuart I. Feldman. Make–a computer program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, 1979.

[Fernandez 95] Mary Fernandez. Simple and Effective Link-time Optimization of Modula-3 Programs. *SIGPLAN Notices*, pages 103–115, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[Fisher 81] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computing*, 30(7):478–490, July 1981.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Lanaguge and its Implementation*. Addision-Wesley, Reading, MA, 1983.

[Gosling et al. 96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[Graham et al. 82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. t gprof:pldi A Call Graph Execution Profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.

[Grove & Torczon 93] Dan Grove and Linda Torczon. Interprocedural Constant Propagation: A Study of Jump Function Implementations. *SIGPLAN Notices*, 28(6):90–99, June 1993. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[Grove 95] David Grove. The Impact of Interprocedural Class Analysis on Optimization. In *Proceedings CASCON '95*, pages 195–203, Toronto, Canada, October 1995.

[Grove et al. 95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA'95 Conference Proceedings*, pages 108–123, Austin, TX, October 1995.

[Hendren et al. 92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis of Imperative Programs. *SIGPLAN Notices*, 27(7):249–260, July 1992. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[Hölzle & Ungar 94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *SIGPLAN Notices*, 29(6):326–336, June 1994. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[Hölzle 94] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, August 1994.

[Hölzle et al. 91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[Hölzle et al. 92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. *SIGPLAN Notices*, 27(7):32–43, July 1992. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[Hudak 86] Paul Hudak. A Semantic Model of Reference Counting and its Abstraction. In *ACM Symposium on LISP and Functional Programming*, pages 351–363, August 1986.

[Hudak et al. 92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell, Version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[Int95] *Pentium Pro Family Programmer's Reference Manual*. Intel Corporation, Inc., Santa Clara, CA, 1995.

[Jones et al. 93] Neil D. Jones, Carstein K. Gomarde, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, NY, 1993.

[Katz & Weise 92] M. Katz and D. Weise. Towards a New Perspective on Partial Evaluation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation '92*, pages 29–36. Yale University, 1992.

[Keppel 91] David Keppel. A Portable Interface for On-the-Fly Instruction Space Modifiction. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–95, Santa Clara, California, 1991.

[Kiczales & Rodriguez 89] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. Technical Report SSL 89-95, Xerox PARC Systems Sciences Laboratory, 1989.

[Kilian 88] Michael F. Kilian. Why Trellis/Owl Runs Fast. Unpublished manuscript, March 1988.

[Knoblock & Ruf 96] Todd B. Knoblock and Erik Ruf. Data Specialization. *SIGPLAN Notices*, pages 1215–225, May 1996. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[Kolte & Wolfe 95] Priyadarshan Kolte and Michael Wolfe. Elimination of Redundant Array Subscript Range Checks. *SIGPLAN Notices*, pages 270–278, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[Kranz et al. 86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: An Optimizing Compiler for Scheme. *SIGPLAN Notices*, 21(7):219–233, July 1986. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*.

[Landi & Ryder 92] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[Larus & Schnarr 95] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. *SIGPLAN Notices*, pages 291–300, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[Lea 90] Doug Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, San Francisco, CA, April 1990.

[Lee & Smith 84] Johnny K. F. Lee and Alan Jay Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 21(7):6–22, January 1984.

[Lim & Stolcke 91] Chu-Cheow Lim and Andreas Stolcke. Sather Language Design and Performance Evaluation. Technical Report TR 91-034, International Computer Science Institute, May 1991.

[Martin et al. 95] Randy Martin, Yung-Chin Chen, and Ken Yeager. *MIPS R10000 Microprocessor User's Manual-Version 1.1*. MIPS Technologies, Inc., Mountain View, CA, 1995. http://www.mips.com/r10k/.

[Meyer 92] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, 1992.

[Milner et al. 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Mowry et al. 92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, Boston, Massachusetts, 1992.

[Mueller & Whalley 95] Frank Mueller and David B. Whalley. Avoiding Conditional Branches by Code Replication. *SIGPLAN Notices*, pages 56–66, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[Myers 95] Andrew C. Myers. Bidirectional Object Layout for Separate Compilation. In *OOPSLA'95 Conference Proceedings*, pages 124–139, Austin, TX, October 1995.

[Nelson 91] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Oxhøj et al. 92] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 329–349, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Palsberg & Schwartzbach 91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings OOPSLA '91*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Pande & Ryder 94] Hemant D. Pande and Barbara G. Ryder. Static Type Determination for C++. In *Proceedings of Sixth USENIX C++ Technical Conference*, 1994.

[Perleberg & Smith 93] Chris Perleberg and Alan Jay Smith. Branch Target Buffer Design and Optimization. *IEEE Transactions on Computing*, 42(4):396–412, April 1993.

[Plevyak & Chien 94] John Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings OOPSLA '94*, pages 324–340, Portland, OR, October 1994.

[Rees & Clinger 86] Jonathan Rees and William Clinger. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12), December 1986.

[Ruf & Weise 91] Erik Ruf and Daniel Weise. Using Types to Avoid Redundant Specialization. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation '91*, pages 321–333. ACM, 1991.

[Schaffert et al. 85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment, Language Reference Manual. Technical Report DEC-TR-372, Digital Equipment Corporation, November 1985.

[Schaffert et al. 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *Proceedings OOPSLA '86*, pages 9–16, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Scheifler 77] Robert W. Scheifler. An Analysis of Inline Substition for a Structured Programming Language. *Communications of the ACM*, 20(9):647–654, September 1977.

[Shao & Appel 95] Zhong Shao and Andrew Appel. A type-based compiler foor Standard ML. *SIGPLAN Notices*, pages 116–129, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

[Shivers 88] Olin Shivers. Control-Flow Analysis in Scheme. *SIGPLAN Notices*, 23(7):164–174, July 1988. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[Srivastava & Eustace 94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Notices*, 29(6):196–205, June 1994. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[Srivastava 92] Amitabh Srivastava. Unreachable Procedures in Object-Oriented Programming. *ACM Letters on Programming Languages and Systems*, 1(4):355–364, December 1992.

[Stallman 90] Richard M. Stallman. *Using and Porting GNU gcc Version 2.0*. Free Software Foundation, Cambridge, MA, 1990.

[Steele 90] Guy L Steele. *Common Lisp: The Language, second edition*. Digital Press, Bedford, MA, 1990.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addision-Wesley, Reading, MA, 1991.

[Sun91] *SPARC Architecture Manual, Version 8*. Sun Microsystems., Mountain View, CA, 1991.

[Tarditi et al. 96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Bob Harper, and Peter Lee. TIL: A Type-Directed Compiler for ML. *SIGPLAN Notices*, pages 181–192, May 1996. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87*, pages 227–242, December 1987.

[Wagner et al. 94] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate Static Estimators for Program Optimization. *SIGPLAN Notices*, 29(6):85–96, June 1994. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[Wall 91] David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. *SIGPLAN Notices*, 26(6):59–70, June 1991. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[Wilson & Lam 95] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. *SIGPLAN Notices*, pages 1–12, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.

Vita

Jeffrey Dean was born in 1968 in Hawaii, and spent his childhood in such varied locales as Honolulu, the Phillipines, Boston, Uganda, Arkansas, Minneapolis, Somalia, and Atlanta (in order). He attended the University of Minnesota from 1986 to 1990, graduating *summa cum laude* with a double major in computer science and economics, despite a lackluster attendance record in many classes. After graduation, he worked for a year for the World Health Organization's Global Programme on AIDS in Geneva, Switzerland. After some year-end accountingrevealed that he had spent the equivalent of $3000 on Coke in a year in Geneva, he decided that he should return to a land with cheaper caffeinated beverages (quite a coke habit, huh?). In 1991, he enrolled in the Ph.D. program at the University of Washington in sunny Seattle, where he spent five years learning about compilers, babies, coffee, and just generally having a great time.

Jeff is still lacking Antarctica, Australia, and South America in his quest to play pickup basketball on every continent.