Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis

Jeffrey Dean, David Grove, and Craig Chambers

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350 USA
{jdean,grove,chambers}@cs.washington.edu

Abstract. Optimizing compilers for object-oriented languages apply static class analysis and other techniques to try to deduce precise information about the possible classes of the receivers of messages; if successful, dynamically-dispatched messages can be replaced with direct procedure calls and potentially further optimized through inline-expansion. By examining the complete inheritance graph of a program, which we call *class hierarchy analysis*, the compiler can improve the quality of static class information and thereby improve run-time performance. In this paper we present class hierarchy analysis and describe techniques for implementing this analysis effectively in both statically- and dynamically-typed languages and also in the presence of multi-methods. We also discuss how class hierarchy analysis can be supported in an interactive programming environment and, to some extent, in the presence of separate compilation. Finally, we assess the bottom-line performance improvement due to class hierarchy analysis alone and in combination with two other "competing" optimizations, profile-guided receiver class prediction and method specialization.

1 Introduction

Object-oriented languages foster the development of reusable, extensible class libraries and frameworks [Johnson 92]. For example, the InterViews graphics framework [Linton *et al.* 89] defines a collection of interacting base classes. The base classes define a set of messages that are to be defined or overridden in subclasses. Clients of the framework specialize it to their use by providing application-specific subclasses of the framework's base classes with the appropriate operations defined. Other frameworks have a similar structure, exploiting inheritance and dynamic binding of messages to make library code customizable and malleable.

Heavy use of inheritance and dynamically-bound messages is likely to make code more extensible and reusable, but it also imposes a significant performance overhead, compared to an equivalent but non-extensible program written in a non-object-oriented manner. In some domains, such as structured graphics packages, the performance cost of the extra flexibility provided by using a heavily object-oriented style is acceptable. However, in other domains, such as basic data structure libraries, numerical computing packages, rendering libraries, and trace-driven simulation frameworks, the cost of message passing can be too great, forcing the programmer to avoid object-oriented programming in the "hot spots" of their application. For example, hybrid languages like C++ [Stroustrup 91], Modula-3 [Nelson 91, Harbison 92], and CLOS [Bobrow *et al.* 88, Gabriel *et al.* 91] provide non-object-oriented built-in array data structures that are more

W. Olthoff (Ed.): ECOOP '95, LNCS 952, pp. 77-101, 1995.

efficient than would be a typical class-based extensible implementation using dynamically-dispatched fetch and store operations, Sather [Omohundro 94, Szypersky *et al.* 93] allows the programmer to explicitly select where subtype polymorphism is allowed, trading away reusability for performance, and it is common practice in C++ programming to avoid virtual function calls along common execution paths, sometimes leading to contorted, hard-to-understand and hard-to-extend code.

Compilers can reduce the cost of dynamically-dispatched messages in a number of ways. For example, *static class analysis* identifies a superset of the set of possible classes of objects that can be stored in variables and returned from expressions. Sometimes class analysis determines that the receiver of a message can be an instance of only one class, allowing the dynamically-dispatched message to be replaced with a direct procedure call (i.e., *statically-bound*) at compile-time and further optimized using inline expansion if the target procedure is small. If static class analysis determines that the receiver of a message can be one of a small set of classes, the dynamically-dispatched message can be replaced with a "type-case" expression, implemented with a series of run-time class tests, each branching to direct procedure calls implementing that case; executing one or two run-time class tests followed by an inlined version of the called procedure can be faster than performing a general run-time method lookup, particularly if additional optimizations of the called and calling methods can take place after inlining. Several other compiler techniques have been investigated for reducing the cost of message passing:

- *Profile-guided receiver class prediction* can support a type-casing-style optimization where static analysis is unable to determine precise information about the receiver of a message. The profile information representing the expected receiver class distribution of particular messages or call sites can be hard-wired into the compiler [Deutsch & Schiffman 84, Chambers *et al.* 89], gathered and exploited on-line [Hölzle & Ungar 94], and/or gathered off-line and exploited via recompilation [Garrett *et al.* 94, Calder & Grunwald 94].
- *Method specialization* can produce faster specialized versions of a method for particular inheriting subclasses; each specialized version can be optimized for the particular class or classes of the receiver for which the method is being specialized. Specializations for a given source method can be produced obliviously for each inheriting subclass [Kilian 88, Chambers & Ungar 89, Lea 90, Lim & Stolcke 91] or they can be produced selectively for groups of inheriting subclasses guided by execution frequency profiles [Dean *et al.* 95].

Class hierarchy analysis is another idea for speeding messages. When the compiler compiles a method, it knows statically that the receiver of the method is some subclass S of the class C containing the method. Unfortunately, in the absence of additional information, the compiler cannot optimize messages sent to the method's receiver, because the subclass S may override any of C's dynamically-dispatched methods. Class hierarchy analysis resolves this dilemma by supplying the compiler with complete knowledge of the program's class inheritance graph and the set of methods defined on each class. In the presence of this global information about the program being compiled, the compiler can infer statically a specific set of possible classes given that the receiver is a subclass of the class C, and messages sent to the method's receiver can be optimized. In particular, if there are no overriding methods in subclasses, a message sent to the method's receiver can be replaced with a direct procedure call and perhaps inlined. This sort of optimization would be especially important in the case of highly-extensible frameworks, where a great deal of flexibility is incorporated in the form of dynamically-dispatched messages within the framework base classes, but where only a limited portion of the potential flexibility is exploited by any particular application. For example, InterViews

supports the display and manipulation of arbitrary graphical shapes, but if a particular application only implements a rectangle concrete subclass, then all the dynamically-dispatched calls within the framework for manipulating arbitrary shapes can be replaced with direct calls to the appropriate rectangle methods.

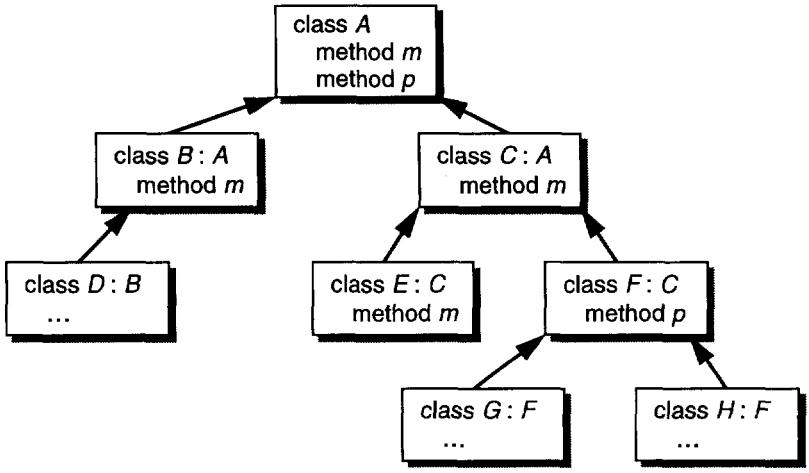
Class hierarchy analysis has long been known informally as a possible optimization among implementors of optimizing compilers for object-oriented languages, but we are unaware of any previous studies of the effectiveness and costs of this technique. Moreover, class hierarchy analysis is just one of a number of candidate optimizations that could be incorporated into an optimizing compiler, and the question remains as to which is the most cost-effective combination to include. In this paper we perform such a study:

- We describe several implementation techniques for efficiently incorporating class hierarchy analysis into a compiler, in particular into an existing static class analysis framework. Our techniques scale to support multi-method-based languages; efficient compile-time method lookup in the presence of multi-methods is substantially harder than for mono-methods.
- We address programming environment concerns of achieving fast turnaround for programming changes and supporting independent development of libraries, which could be adversely affected by a whole-program analysis such as class hierarchy analysis.
- We measure the run-time performance benefit and compile-time cost of class hierarchy analysis on several large programs written in Cecil [Chambers 92, Chambers 93], a pure object-oriented language with multi-methods. Moreover, we also measure the run-time performance benefits and compile-time costs of profile-guided receiver class prediction and method specialization separately and in combination with class hierarchy analysis.

The next section of this paper describes our integration of class hierarchy analysis into static class analysis and addresses programming environment concerns. Section 3 reports on our experimental evaluation of class hierarchy analysis, profile-guided receiver class prediction, and method specialization. Section 4 describes some related work and Section 5 offers some conclusions.

2 Class Hierarchy Analysis

By exploiting information about the structure of the class inheritance graph, including where methods are defined (but not depending on the implementation of any method nor on the instance variables of the classes), the compiler can gain valuable static information about the possible classes of the receiver of each method being compiled. To illustrate, consider the following class hierarchy:



Consider the situation where the method *p* in the class *F* contains a send of the *m* message to self. *m* is declared to be a virtual function (there are several implementations of *m* for subclasses of *A*, and the right implementation should be selected dynamically). As a result, with only static intraprocedural class analysis 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, or the like. 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*. Class hierarchy analysis is a direct method for determining this without programmer intervention.

2.1 Alternatives to Class Hierarchy Analysis

Other languages have alternative approaches for achieving a similar effect. In C++ a programmer can declare whether or not a method is virtual (methods default to being non-virtual). 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:

- The C++ programmer must make explicit decisions of which methods need to be virtual, making the programming process more difficult. When developing a reusable framework, the framework designer must make decisions about which

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.

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 goes unused for any particular application, incurring an unnecessary run-time performance cost. 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 which clients may not be able to change. Class hierarchy analysis, as an automatic mechanism, 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 particularly subtree of the inheritance graph, there will 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 implemented with a dynamically-bound message, but 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. In particular, it is always the case that a message sent to the receiver of a method defined in a leaf class will have only one target implementation and hence can be implemented as a direct procedure call, regardless of whether or not the target method is declared virtual. For the benchmark programs discussed in Section 3, slightly more than half of the message sends that were statically bound through class hierarchy analysis could not have been made non-virtual in C++ (i.e., had more than a single definition of the routine).

In a similar vein, Trellis [Schaffert *et al.* 85, Schaffert *et al.* 86] allows a class to be declared with the no_subtypes annotation and Dylan [Dyl92] allows a class to be sealed, both of which inform the compiler that no subclasses exist. These annotations allow the compiler to treat the class as a leaf class and compile all messages sent to objects statically known to be of the class as direct procedure calls. Sealing has similar weaknesses relative to class hierarchy analysis as do non-virtual functions in C++: programmers have to predict in advance, in the source code, which classes are to be sealed, and opportunities for static binding will be missed, relative to class hierarchy analysis, when a class has unknown subclasses but none of the subclasses override certain methods.

2.2 Implementation

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; the compiler uses this information 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]:

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 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.

2.2.1 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:

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. (This framework for representing static class analysis information is similar to Palsberg and Schwartzbach's static type system [Palsberg & Schwartzbach 94].) Cones tend to be concise summaries of sets of classes: in our implementation, when compiling a 52,000-line benchmark program with 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 integration is crucial to adapting techniques developed for dynamically-typed object-oriented languages to work effectively for 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; CLOS takes a similar view on integrating the standard Lisp data types with user-defined classes.

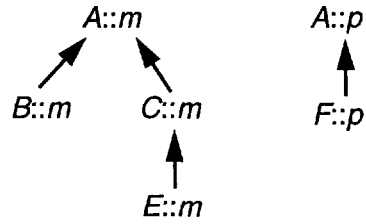
2.2.2 Method Applies-To Sets

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, $C::m$. Our measurements 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 could be slow for large sets (e.g., cones of classes with many subclasses).

We have pursued an alternative approach that compares whole sets of classes at once. We first precompute for each method the set of classes for which that method is the appropriate target; we call this set the *applies-to* set. (In our compiler, we compute the applies-to sets of methods 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 program, this cache contained 7,686 entries, and a total of 54,211 queries of the cache were made during compilation.)

The efficiency of this approach to compile-time method lookup 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 iff M_1 overrides M_2. For the running example, we construct the following partial order:



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 subtracting 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 is $O(N^2)$,[1] but 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(C1) overlaps Class(C2) iff C1 = C2 or C2 inherits from C1. Overlaps testing of arbitrary Difference sets is complex and can be expensive. Since applies-to sets in general are Differences, overlaps testing of a receiver class set against a collection of applies-to Difference sets could be expensive. To represent irregular applies-to sets more efficiently, we convert Difference sets into a flattened BitSet representation. 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 called with objects other than those in the method's applies-to set; the applies-to set only applies for normal dynamically-dispatched message 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.

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

2.2.3 Support for Dynamically-Typed Languages

In a dynamically-typed language, there is the possibility 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 root class, 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.

Knowledge of the class hierarchy and the location of defined methods can improve the results of receiver class prediction, a common technique used when the available static class information is not precise enough to lead to static binding of a message send. If the compiler can predict the expected class(es) of the message's receiver, either based on the name of the message and a hard-wired table in the compiler (as in Smalltalk-80 and the Self-91 system) or on dynamic profile data (as in the Self-93 system and Cecil), then it can insert run-time class tests for the expected classes. The compiler generates a full message send to handle any unexpected classes that occur at run-time:

Before Class Prediction:

```
a := s.area();
```

After Class Prediction:

```
if (s.class == Rectangle) {
    // statically bind to rectangle's area; inline if small
    a := s.Rectangle::area();
} else if (s.class == Circle) {
    // statically bind to circle's area; inline if small
    a := s.Circle::area();
} else {
    // a full message send to handle unexpected cases
    a := s.area();
}
```

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. (In statically-typed languages, using class hierarchy information to convert static type declarations into Cone class set representations accomplishes a similar purpose.) 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. 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.) In the absence of class hierarchy information, the compiler must assume that some other class could implement the area message (or,

in a statically-typed language, that some other class could be a subtype of the Shape static type), and consequently include support for the third "unexpected" case. When compiling our 52,000-line benchmark program, elimination of unexpected cases using class hierarchy analysis occurred 3,232 times; 3,004 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.

2.2.4 Support for Multi-Methods

The above strategy for static class analysis in the presence of class hierarchy analysis and/or static type declarations works for singly-dispatched languages with one message receiver, but it does not support languages with multi-methods, such as CLOS, Dylan, and Cecil. 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.[1] To represent many common sets of tuples of 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 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 is modified to support multi-methods as follows. For each method, we precompute the method's applies-to *tuple* of class sets; this tuple 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 $<\text{Cone}(C_1), ..., \text{Cone}(C_k)>$. 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.

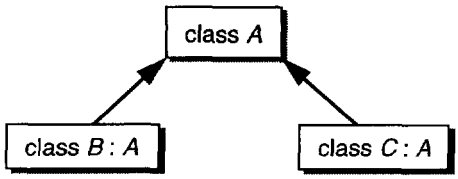
Efficient multi-method static class analysis relies on efficient overlaps testing and difference operations on tuples. Testing whether one tuple overlaps another is straightforward: each element class set of one tuple must overlap the corresponding class set of the other tuple. Computing the difference 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 straightforward and correct representation would be a union of k k-tuples, where each tuple has one element class set difference taken:

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

If the $S_i - T_i$ element set is empty, then the i-th k-tuple is dropped from the union: 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 optimizations are important in practice.

1. We assume that the compiler can determine statically which subset of a message's arguments can 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 compiler examines all methods with the same name and number of arguments and finds 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 a single argument.

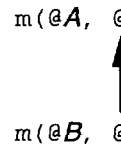
For example, consider the following class hierarchy and multi-methods (x@*X* is the syntax we use for indicating that the x formal argument of a multi-method is specialized for the class *X*):




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

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

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



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

m(@*A*, @*A*, @*A*): $\langle\{A,C\}, \{A,B,C\}, \{A,B,C\}\rangle \cup \langle\{A,B,C\}, \{A,B\}, \{A,B,C\}\rangle$

m(@*B*, @*C*, @*A*): $\langle\{B\}, \{C\}, \{A,B,C\}\rangle$

(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:

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

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

m(@*A*, @*A*, @*A*): $\langle\{A\}, \{A,B,C\}, \{A,B,C\}\rangle \cup \langle\{A,C\}, \{A,C\}, \{A,B,C\}\rangle \cup$
$\langle\{A,C\}, \{A,B,C\}, \{A,B\}\rangle \cup \langle\{A,B\}, \{A,B\}, \{A,B,C\}\rangle \cup$
$\langle\{A,B,C\}, \{A\}, \{A,B,C\}\rangle \cup \langle\{A,B,C\}, \{A,B\}, \{A,B\}\rangle$

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:

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

By taking the intersection of the first i-1 elements of the ith 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*): $\langle\{A\}, \{A,B,C\}, \{A,B,C\}\rangle \cup \langle\{C\}, \{A,C\}, \{A,B,C\}\rangle \cup$
$\langle\{C\}, \{B\}, \{A,B\}\rangle \cup \langle\{B\}, \{A,B\}, \{A,B,C\}\rangle$

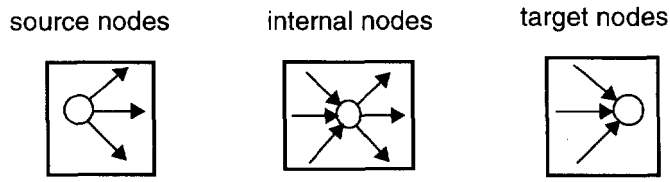
As a final guard against exponential growth, we impose a limit on the number of class set terms in the resulting tuple representation, beyond which we stop narrowing (through subtraction) a method's applies-to set. We rarely resort to this final ad hoc measure: when compiling a 52,000-line Cecil program, 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, using the simpler representation described first, many applies-to sets would have exceeded the 64-term threshold.

2.3 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.

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. In previous work, we have developed a framework for maintaining intermodule dependency information [Chambers *et al.* 95]. This framework is effective at representing the compilation dependencies introduced by class hierarchy analysis.

In the dependency framework, intermodule dependencies are represented by a directed, acyclic graph structure. Nodes in this graph represent information, including pieces of the program's source and information resulting from various interprocedural analyses such as class hierarchy analysis, 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. Depending on the number of incoming and outgoing edges, we classify nodes into three categories:

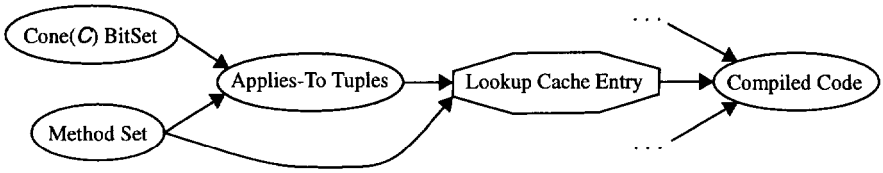


- *Source nodes* have only outgoing dependency edges. They represent information present in the source modules comprising the program, such as the source code of procedures and the class inheritance hierarchy.
- *Target nodes* have only incoming dependency edges. They represent information that is an end product of compilation, such as compiled . o files.
- *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.

The dependency graph is constructed incrementally during compilation. Whenever a portion of the compilation process uses a piece of information that could change, the compiler adds an edge to the dependency graph from the node representing the

information used to the node representing the client of the information. When changes are made to the source program, the compiler computes what source dependency nodes have been affected and propagates invalidations downstream from these nodes. This invalidates all information (including compiled code modules) that depended on the changed source information.

In our compiler, static class analysis queries a compile-time method lookup cache to attempt to determine the outcome of message lookups statically; this cache is indexed with a message name and a tuple of argument class sets and returns the set of methods that might be called by such a message. To compute an entry in the method lookup cache, the compiler tests the applies-to tuples of methods with a matching name, in turn examining the BitSet representation of the set of classes represented by a Cone class set, which was computed from the underlying class inheritance graph. To support selective recompilation of optimized code, dependency graph nodes are introduced to model information derived from the source code:



- one kind of dependency node represents the BitSet representation of the set of subclasses of a class (one product of class hierarchy analysis),
- another kind of dependency node represents the set of methods with a particular name (another product of class hierarchy analysis),
- a third kind of dependency node represents the applies-to tuples of the methods, which is derived from the previous two pieces of information, and
- a fourth kind of dependency node guards each entry in the compile-time method lookup cache.

If the set of subclasses of a given class is changed or if the set of methods with a particular name and argument count is changed, the corresponding source dependency nodes are invalidated. This causes all downstream dependency nodes to be invalidated recursively, eventually leading to the appropriate compiled code being invalidated and subsequently recompiled.

To support greater selectivity and avoid unnecessarily invalidating any compiled code, some of the internal nodes in the dependency framework are *filtering nodes*. When invalidated, a filtering node will first check whether the information it represents really has changed; only if the information it represents has changed will a filtering node invalidate its successor dependency nodes. The compile-time method lookup cache entries are guarded by such filtering nodes. If part of the inheritance graph is changed or a new method is added, then downstream method lookup results *may* have changed, but often the source changes do not affect all potentially dependent method lookup cache entries. By rechecking the method lookup when invalidated, and squashing the invalidation if the method lookup outcome was unaffected by a particular source change, many unnecessary recompilations are avoided.

Empirical evaluation using a trace of a month's worth of actual program development indicates that the dependency-graph-based approach reduces the amount of recompilation required during incremental compiles by a factor of seven over a coarser-grained C++-style header file scheme, in the presence of class hierarchy analysis, and by a factor of two

over the Self compiler's previous state-of-the-art fine-grained dependency mechanism [Chambers & Ungar 91]. Of course, more recompilation occurs in the presence of class hierarchy analysis than would occur without it, but for these traces the number of files recompiled after a programming change is often no more than the number of files directly modified by the changes. A more important concern with our current implementation is that many filtering nodes may need to be checked after some programming changes, and even if few compiled files are invalidated, a fair amount of compilation time is expended in checking caches. The size of the dependency graph is about half as large as the executable for the program being compiled, which is acceptable in our program development environment; coarser-grained dependency graphs could be devised that save space at the cost of reduced selectivity. Further details are available elsewhere [Chambers *et al.* 95].

2.4 Optimization of Incomplete Programs

Class hierarchy analysis 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; 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 where 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 won't be able to be inlined. Static binding alone still provides significant performance improvements, particularly on modern RISC processors, where dynamically-dispatched message send implementations stall the hardware pipeline. Furthermore, some optimizing linkers are able to optimize static calls by inlining the target routine's machine code at link time [Fernandez 95], although the resulting code is not as optimized as what could be done in the compiler.

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. In previous work, we have developed 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 [Dean *et al.* 95]. 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 50% or more with only a 5% to 10% compiled code space increase, although we would expect a larger relative space overhead if the algorithm were applied to a library in isolation.

In summary, although class hierarchy analysis is most effective when the whole program is available, it can still be applied in situations where only portions of the program are available. Using the techniques described above, it can be applied to incomplete programs and to libraries independent of client applications, at some cost in missed optimization opportunities and/or increased compiled code space.

3 Empirical Assessment

Class hierarchy analysis, method specialization, and profile-guided receiver class prediction are all techniques for increasing the amount of class information available to the optimizer at compile time. All three represent different, and partially overlapping, approaches to solving the same fundamental problem: enabling the static binding of dynamic dispatches. In this section, we examine the effectiveness of these three approaches in isolation and in combination, focusing on the following questions:

- What is the impact of class hierarchy analysis on program performance?
- How effective is class hierarchy analysis in comparison to specialization? Can additional benefit be gained from combining class hierarchy analysis and specialization?
- How much benefit does class hierarchy analysis confer to a system that already performs profile-guided receiver class prediction?

We examine these issues in the context of the Vortex optimizing compiler for Cecil, a pure object-oriented language with multi-methods. Table 1 describes the five medium-to-large Cecil programs that we used as benchmarks. Appendix A includes the raw performance data.

Table 1: Cecil Benchmarks

Program	Lines[a]	Description
Richards (Rich)	400	Operating systems simulation
Deltablue (Delta)	650	Incremental constraint solver
InstrSched (Instr)	2,400	MIPS global instruction scheduler
Typechecker (Type)	17,000[b]	Cecil static typechecker
Compiler (Comp)	43,800[b]	Vortex optimizing compiler

a. Not including 8,500-line standard library.

b. The typechecker and compiler share approximately 12,000 lines of code.

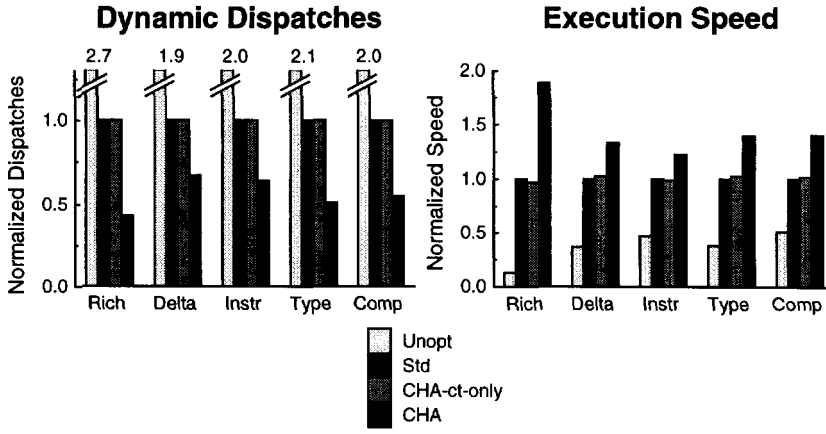


Figure 1: Number of dynamic dispatches and execution speed

3.1 Effectiveness of Class Hierarchy Analysis

Since class hierarchy analysis provides the compiler with additional information about the classes of program variables (in particular the receiver(s) of the message being compiled), we would expect that the compiler would be able to statically bind more dynamic dispatches. Additionally, as discussed in Section 2.2.3, in some situations the compiler can determine when unexpected cases of a message send are guaranteed to fail, thus improving the quality of static analysis downstream of the send.

To measure the impact of class hierarchy analysis, we compiled the benchmark programs using the following set of compiler optimizations:

- ***unopt:*** No optimizations.
- ***std***: Standard static intraprocedural analyses, including iterative intraprocedural class analysis, inlining, hard-wired class prediction for a small set of common messages, closure optimizations, extended splitting [Chambers & Ungar 90] and other standard intraprocedural optimizations such as CSE, constant folding and propagation, and dead code elimination.
- ***cha-ct-only***: Standard (*std*) augmented by a limited usage of class hierarchy analysis. The results of class hierarchy analysis are used only to optimize the uncommon cases after run-time class tests, as described in Section 2.2.3.
- ***cha***: Standard augmented by class hierarchy analysis. Class hierarchy analysis is used to provide class information about the receiver(s) of a method and to determine when messages sends are guaranteed to fail.

Figure 1 shows the dynamic number of dynamic dispatches and the execution speeds of the benchmark programs, normalized to those of *std.* Augmenting standard intraprocedural techniques with class hierarchy analysis resulted in a 23% to 89% improvement in execution speed for these applications. Since the performance difference between *std* and *cha-ct-only* was negligible, we can conclude that almost all of the runtime benefits seen in *cha* are due to additional receiver class information and not from optimizing the unexpected branches of class tests inserted by hard-wired class prediction, therefore we expect the improvements due to class hierarchy analysis to be significant even in object-oriented languages lacking used-defined control structures.

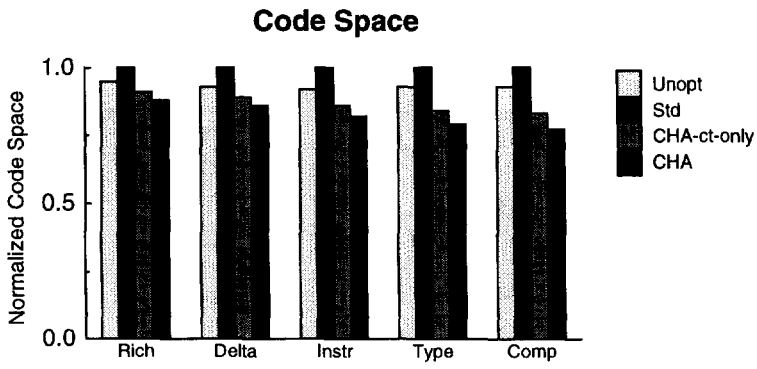


Figure 2: Compiled Code Space

In addition to improving execution speed, class hierarchy analysis reduces compiled code size, as shown in figure 2. Executables compiled with class hierarchy analysis were 12% to 21% smaller than *std* executables. By comparing the relative heights of the bars we can see that most of this reduction in code space was due to the replacement of call sites which are guaranteed to result in a message-not-understood error by a simple call to an error routine. Such call sites mainly occurred in the off branches of class tests inserted by hard-wired class prediction and thus, one would expect that class hierarchy analysis would have a smaller impact on compiled code space in languages without user-defined control structures.

3.2 Class Hierarchy Analysis and Specialization

Method specialization creates multiple copies of a single source method, each one of which is compiled with more precise static class information about the method receiver(s) thus enabling static binding and inlining of messages sent to self. Class hierarchy analysis makes a similar contribution. In some sense, specialization and class hierarchy analysis are competing approaches to gaining the same sort of information. An important question, then, is "what are the relative benefits and costs of the two techniques?" Since a specialized method has exact class information about the receiver(s) of the method, we would expect that specialization would yield better results than class hierarchy analysis, but specialization accrues its benefits at the cost of increased compiled code space. In this section, we examine the impact of class hierarchy analysis and method specialization, both in isolation and in combination, using the following set of compiler optimizations:

- ***std***: Standard static intraprocedural analyses, as described in Section 3.1.
- ***cha***: Standard augmented by class hierarchy analysis.
- ***cust-k***: Standard augmented by the *customization* form of method specialization. Customization is the specialization strategy used in Self, Trellis, and Sather implementations where a specialized version of a method is generated for each inheriting subclass. We extended customization to handle multi-methods by specializing on all combinations of subclasses of the dispatched arguments.[1]

1. We used profile data to determine which of the specializations produced by *cust-k* actually needed to be generated, since it was infeasible to actually compile them all. In effect, this simulates Self-style dynamic compilation.

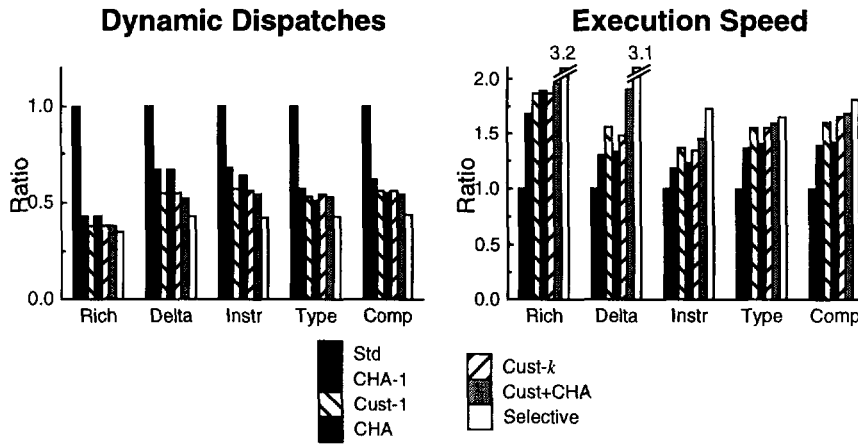


Figure 3: Number of dynamic dispatches and execution speed

- ***cust-1***: Standard augmented by customization on only the first receiver. Customization on only the first receiver is more feasible in practice than customizing on all combinations of receivers in the case of multi-methods.
- ***cha-1***: Standard augmented by class hierarchy analysis for only the first receiver, to compare against *cust-1*.
- ***cust+cha***: Standard augmented with both *cust-k* and class hierarchy analysis.
- ***selective***: Standard augmented with class hierarchy analysis and a selective specialization algorithm that combines a profile-derived weighted call graph and class hierarchy analysis to select candidates for specialization [Dean *et al.* 95].

Figure 3 shows the normalized number of dynamic dispatches and execution speeds for the benchmark programs compiled with these seven configurations. As expected, in most cases customization had a larger impact than class hierarchy analysis, speeding up programs by 48% to 87%. Combining class hierarchy analysis and naive customization (*cust+cha*) yielded only small additional benefits. However, the runtime benefits of customization came at the cost of a large increase in compiled code space and compile time. *Cust-1* executables were 2.5 to 3.5 times larger than *std* executables and *cust-k* executables are too large to actually be built using standard static compilation techniques; even in a system with dynamic compilation, *cust-k* would require compiling roughly 1.5 to 2 times as many methods as *std*. In contrast, *cha* executables were 12% to 21% smaller than *std*. By far, the best results were achieved by *selective*, which increased execution speed by 52% to 210% while increasing compiled code space by only 4% to 10%.

3.3 Class Hierarchy Analysis and Profile-Guided Receiver Class Prediction

Profile-guided receiver class prediction has been shown to substantially improve the performance of applications written in pure object-oriented languages. It is unclear whether or not adding class hierarchy analysis to a system that already performs profile-guided receiver class prediction would result in any significant improvements [Hölzle 94]. To examine this question, we utilized the following compiler configurations:

- ***std***: Standard intraprocedural optimizations as described in Section 3.1.
- ***cha***: Standard augmented by class hierarchy analysis

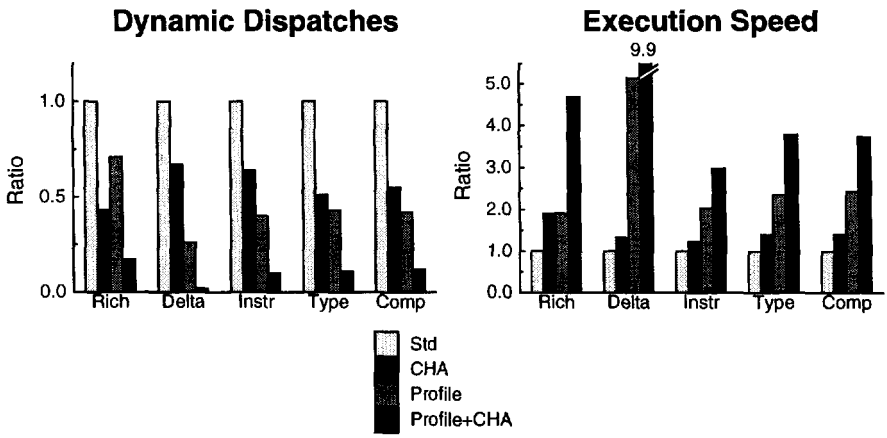


Figure 4: Number of dynamic dispatches and execution speed

- ***profile***: Standard augmented with profile-guided receiver class prediction.
- ***profile+cha***: Standard augmented with profile-guided receiver class prediction and class hierarchy analysis.

Figure 4 shows the normalized number of dynamic dispatches and execution speeds of the applications. These numbers show that profile-guided receiver class prediction is the most effective of the three techniques in isolation, improving application execution speeds by 90% to 410% over *std*. However, augmenting profile-guided receiver class prediction with class hierarchy analysis (*profile+cha*) yielded surprisingly large additional improvements of 45% to 125% over *profile* alone. Part of this improvement can be explained by a reduction in the number of call sites at which the compiler, due to a lack of sufficient static class information, is forced to fall back on profile information and insert explicit class tests. However, the improvements can not solely be explained by this effect, since combining the two optimizations yields larger benefits than can be explained even by assuming that their benefits in isolation are completely additive. For example, in the Compiler benchmark, *cha* is 41% faster than *std* and *profile* is 142% faster than *std*. Multiplying these two speedups results in a projected speedup of 241% over *std*, which is smaller than the observed speedup of 272%. By examining the number of dynamic dispatches, we can see that it is not the case that *profile+cha* is able to eliminate more dispatches than the sum of *profile* and *cha* (in the Compiler benchmark, *cha* eliminated 45% of the dynamic dispatches present in *std* and *profile* eliminated 58%, but *profile+cha* only eliminated 88%). We believe that the large speedups observed in *profile+cha* are due to the increased effectiveness of other compiler optimizations, such as CSE and register allocation, that was enabled by the simplified control flow graphs produced by this configuration.

In our implementation of profile-guided receiver class prediction, we only insert class tests if the target method is a desirable candidate for inlining; if the target method is not going to be inlined, then we leave the call site unchanged. In the Richards benchmark, several of the frequently-called methods are too large to be inlined, but can be statically-bound by class hierarchy analysis. This explains why *profile* eliminated fewer dynamic dispatches than *cha* for the Richards benchmark.

4 Other Related Work

An alternative to performing whole-program optimizations such as class hierarchy analysis at compile-time is to perform optimizations at link-time. 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. An advantage of performing optimizations at link-time is that, because the optimizations operate on machine code, they can be applied to the whole program, including libraries for which source code is unavailable. However, there are two drawbacks of link-time optimizations. First, because the conversion of message sends to statically-bound calls happens in the linker, rather than the compiler, the compiler's optimization tools cannot be brought to bear on the now statically-bound call site; the indirect benefits of post-inlining optimizations in the compiler can be more important than the direct benefit of eliminating procedure call/return sequences. Second, care must be taken to prevent linking from becoming a bottleneck in the edit-compile-debug cycle. For example, Fernandez's link-time optimizer for Modula-3 performs code generation from a machine-independent intermediate representation for the entire program at every link; Fernandez acknowledges that this design penalizes turnaround time for small programming changes. Additional link-time optimizations would only increase this penalty. In contrast, class hierarchy analysis coupled with a selective invalidation mechanism supports incremental recompilation, fast linking, and compile-time optimization of call sites where source code of target methods is available.

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, Agesen *et al.* 93, Plevyak & Chien 94]. 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 Section 3. Agesen and Hölzle have compared the effectiveness of profile-guided receiver class prediction alone to interprocedural class analysis alone for a suite of small-to-medium sized Self programs, but they did not report on the effectiveness of combining the two techniques [Agesen & Hölzle 95].

5 Conclusions

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 sealed classes in Dylan, 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. Class hierarchy analysis imposes some requirements on the underlying programming environment, particularly to support incremental compilation, but these costs appear to be manageable in practice. These techniques have been implemented in the Vortex compiler for Cecil since the Spring of 1994. In this compiler class hierarchy analysis is always performed as a matter of course, and intermodule dependency links support selective recompilation. The Vortex compiler is itself a 52,000-line Cecil program, undergoing rapid continuous development and extension, providing some evidence that class hierarchy analysis is compatible with a program development environment.

Class hierarchy analysis is only one of a number of optimizations proposed for object-oriented languages; others include method specialization and profile-guided receiver class prediction. We implemented and measured these techniques separately and in combination, on a collection of medium-to-large Cecil programs, to try to determine which techniques are most effective and where the techniques could profitably be combined. Of the techniques that we examined, profile-guided class prediction was the most effective in isolation at improving program performance. However, performing class hierarchy analysis in addition to profile-guided class prediction provided substantial performance improvements over profile-guided class prediction alone. Class hierarchy analysis consumed far less compiled code space than customization, but with smaller performance gains; the best results are achieved by a profile-guided selective specialization algorithm integrated with class hierarchy analysis.

There are two interesting future directions for this work. First, it would be very interesting to extend this performance study to include interprocedural static class analysis algorithms as they mature. Second, the effectiveness of these techniques in pure object-oriented languages like Cecil has been demonstrated, but their effectiveness and relative value when applied to hybrid, statically-typed object-oriented languages such as C++ and Modula-3 remains an open question. We are in the process of porting compiler front-ends for C++ and Modula-3 to the Vortex optimizing compiler back-end in order to be able to perform such experiments.

Acknowledgments

We thank William Pugh for his help in devising a more efficient tuple difference implementation. Charles Garrett performed many of the initial studies of profile-guided receiver class prediction. Vitaly Shmatikov, Ben Teitelbaum, Tina Wong, and Matthew Parker have contributed to the Cecil implementation.

This research is supported in part by an NSF Research Initiation Award (contract number CCR-9210990), an NSF Young Investigator Award (contract number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM, and Pure Software. Stephen North and Eleftherios Koutsofios of AT&T Bell Laboratories provided us with `dot`, a program for automatic graph layout.

Other papers on the Cecil programming language and the Vortex optimizing compiler are available via anonymous `ftp` from cs.washington.edu:/pub/chambers and via the *World Wide Web URL http://www.cs.washington.edu/research/projects/cecil.*

References

[Agesen & Hölzle 95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages. Technical Report TRCS 95-04, Department of Computer Science, University of California, Santa Barbara, March 1995.

[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.

[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.

[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.

[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. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[Chambers 93] 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 *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.

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

[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.

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

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

[Gabriel *et al.* 91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9):28–38, September 1991.

[Garrett *et al.* 94] Charlie Garrett, Jeffrey Dean, David Grove, and Craig Chambers. Measurement and Application of Dynamic Receiver Class Distributions. Technical Report UW-CS 94-03-05, University of Washington, March 1994.

[Harbison 92] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[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.

[Johnson 92] Ralph Johnson. Documenting Frameworks Using Patterns. In *Proceedings OOPSLA '92*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

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

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

[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.

[Linton *et al.* 89] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, 2(2):8–22, February 1989.

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

[Omohundro 94] Stephen Omohundro. The Sather 1.0 Specification. Unpublished manuscript from International Computer Science Institute, Berkeley, CA, 1994.

[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, July 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.

[Palsberg & Schwartzbach 94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[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, Oregon, October 1994.

[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.

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

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

[Szypersky *et al.* 93] Clemens Szypersky, Stephen Omohundro, and Stephan Murerzw. Engineering a Programming Language: The Type and Class System of Sather. Technical Report 93-064, International Computer Science Institute, Berkeley, CA, 1993.

Appendix A Raw Data

Table 2: Execution Times (seconds)

Configuration	Richards	Deltablue	InstSched	Typechecker	Compiler
unopt	13.410	31.830	25.080	294	2314
std	1.780	11.880	11.870	113	1176
cha-ct-only	1.830	11.490	11.960	110	1152
cha	0.940	8.940	9.670	81	834
cust-1	0.950	7.620	8.710	73	735
cha-1	1.060	9.160	10.030	83	852
cust-*k*	0.950	8.050	8.870	73	712
cust-*k*+cha	0.910	6.220	8.160	71	700
selective	0.560	3.810	6.850	68	650
profile	0.930	2.310	5.830	48	484
profile+cha	0.380	1.200	4.000	40	316

Table 3: Dynamically Dispatched Message Sends (x1000)

Configuration	Richards	Deltablue	InstSched	Typechecker	Compiler
unopt	10,006	13,461	7,505	75,053	470,488
std	3,697	6,980	4,676	36,131	231,659
cha-ct-only	3,697	6,980	4,664	36,028	231,529
cha	1,583	4,668	3,004	18,566	128,400
cust-1	1,414	3,874	2,655	19,301	129,729
cha-1	1,583	4,668	3,192	20,632	143,578
cust-*k*	1,414	3,874	2,637	19,341	129,701
cust-*k*+cha	1,414	3,624	2,516	19,156	125,095
selective	1,278	3,021	1,969	15,478	101,929
profile	619	159	1,890	15,426	98,435
profile+cha	545	157	474	3,902	28,640