Whole-Program Optimization of Object-Oriented Languages

Craig Chambers, Jeffrey Dean, and David Grove

Department of Computer Science and Engineering, Box 352350
University of Washington
Seattle, Washington 98195-2350 USA

Technical Report 96-06-02
June 1996

{chambers,jdean,grove}@cs.washington.edu
(206) 685-2094; fax: (206) 543-2969

Whole-Program Optimization of Object-Oriented Languages

Craig Chambers, Jeffrey Dean, and David Grove

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, Washington 98195-2350
(206) 685-2094; fax: (206) 543-2969
{chambers,jdean,grove}@cs.washington.edu

Abstract

We describe Vortex, an optimizing compiler intended to produce high-quality code for programs written in a heavily-object-oriented style. To achieve this end, Vortex includes a number of intra- and interprocedural static analyses that can exploit knowledge about the whole program being compiled, including intraprocedural class analysis, class hierarchy analysis, and exhaustive class testing, and profile-guided optimizations such as receiver class prediction and selective specialization. To make whole-program optimization practical, Vortex automatically tracks cross-file optimization dependencies at a fine granularity, triggering selective recompilation of affected compiled files whenever the source program changes. Empirical measurements of five purely object-oriented benchmark programs written in Cecil, ranging in size from several hundred to 75,000 lines of source code, indicate that these optimization techniques improve performance of large programs by more than a factor of three over a system with only intraprocedural static optimizations. Vortex is written in Cecil, and it has been used as its own compiler and optimizer during its development for the past two years. Vortex's optimizations and implementation techniques should be useful for any language or program where optimizations to reduce the cost of polymorphism are important, including object-oriented languages (we are currently adding front-ends for C++, Modula-3, and Java to Vortex to study its effectiveness on these other language styles) and other high-level symbolic, functional, and logic languages.

1 Introduction

Object-oriented programming languages include several features that can be used to make programs more reusable and extensible: *classes* support the construction of encapsulated abstract data types, *inheritance* allows one abstract data type to be implemented as an incremental modification of one or more existing abstract data type implementations, and *dynamic dispatching* (also known as message passing) introduces a level of indirection between client and implementation that allows a single piece of client code to manipulate multiple different implementations, often at different times within the same program run, as long as the implementations support the interface required by the client. Additionally, the presence of these specific features encourages new programming styles and idioms. For example, rich inheritance hierarchies of classes have been developed for particular domains, including standard data structure libraries, graphics packages, and numerical libraries. These libraries often include many related abstractions, since inheritance allows several similar abstractions to be developed and maintained with relatively little effort. The combination of inheritance and dynamic dispatching allows common code patterns to be factored out of specific concrete classes into shared abstract superclasses; dynamic dispatching allows subclass-specific behavior to be invoked from within the shared factored code. And libraries are used by client applications in new ways, such as expecting client applications to add application-specific subclasses to "instantiate" and refine a generic class library; such collections of abstract classes designed for application-specific subclassing are often called frameworks. Object-oriented systems have achieved a remarkable success and popularity, due in large part to the increases in programmer productivity and the natural match of many object-oriented notions to the ways people think about programs and the application domain.

Unfortunately, the new styles of programming fostered by the object-oriented programming language features tend to hinder efficient execution. Dynamic dispatching incurs the direct cost of determining the appropriate routine to execute (based on the dynamic class(es) of the argument(s) of the call), and also the indirect opportunity cost due to preventing inline expansion and post-inlining intraprocedural optimizations. The programming styles fostered by object-oriented programming serve to exacerbate this cost: classes and encapsulation encourage smaller routines with greater frequency of procedure calls, inheritance and factoring encourage increased use of dynamic dispatching, and frameworks designed for application customization through subclassing rely on dynamic dispatching to interact with the new classes. It is a sad fact that the greater the potential benefits of object-oriented programming, the greater is the run-time performance cost. Pure object-oriented languages such as Smalltalk represent an extreme position in this trade-off: all language features are expressed in object-oriented terms, thus preserving maximum flexibility and potential for extension but incurring the greatest run-time performance cost.

We are developing a new optimizing compiler infrastructure for object-oriented languages, named Vortex. Vortex attempts to reduce the run-time performance cost of heavily-object-oriented programming styles through a collection of static analysis-based and dynamic profile-guided techniques culminating with the idea of *whole-program analysis and optimization*: by examining the static and dynamic behavior of the entire application, the compiler attempts to identify where potential flexibility is not needed by the particular application and then choose the most efficient implementation strategy for the flexibility that is needed. Vortex includes the following techniques:

- *Static intraprocedural class analysis* is a kind of dataflow analysis that computes an upper bound on the set of classes whose instances can be stored in variables or result from expression evaluations. Given this information about the receivers of messages, the compiler tries to perform *compile-time method lookup* to replace a dynamically-dispatched message with a statically-bound direct procedure call. *Automatic procedure inlining* serves to further reduce the overhead of statically-bound calls to small procedures. Similar intraprocedural analyses reduce the cost of *closures*, especially when they are used to implement user-defined control structures.
- *Class hierarchy analysis* extends static intraprocedural analysis with knowledge of the entire program's class inheritance graph. Given this knowledge, the compiler can replace knowledge that a variable holds some unknown subclass of a class with more specific knowledge that the variable holds an instance of one of a fixed set of known classes. Class hierarchy analysis is particularly helpful in identifying messages whose default target method is not overridden, i.e., where some potential flexibility is not needed by the particular application being compiled. For languages with multi-methods, knowledge of which formal argument positions are specialized by multi-methods comprising the program enables compile-time method lookup and run-time dispatching to be implemented more efficiently. When class hierarchy analysis is only able to narrow the set of potentially-invoked methods to a few methods, *exhaustive class testing* can be used to insert a few run-time subtyping tests to replace the dynamically-dispatched message with a group of statically-bound calls or inlined code.
- *Dynamic profile information* about the relative frequencies of the classes of receivers at particular call sites is used by Vortex to augment information derived through static analysis. Vortex applies dynamic receiver class frequency information to guide *receiver class prediction*, a technique where common

receiver classes are special-cased at call sites by inserting run-time tests for the expected classes which branch to inlined versions of the corresponding target methods. The *splitting* technique reduces the cost of redundant class tests by replicating paths through the intraprocedural control flow graph in order to preserve intraprocedural class information derived from class tests as long as needed.

- *Selective method specialization* produces multiple versions of a particular source method, each version compiled and optimized for a subset of the original method's potential argument classes. Class hierarchy information is used to identify the set of classes that can share a particular specialized version, and dynamic profile information is used to determine which of the many potential specialized versions of methods are most profitable to generate.

Since whole-program analysis is a form of interprocedural analysis, it can introduce compilation dependencies across modules: if one module is changed, several "independent" modules may become out-of-date and need recompilation. Vortex includes an automatic dependency mechanism to track these cross-module dependencies at a fine grain in an effort to minimize the number of modules that need recompilation after a source program change. In this manner, Vortex compensates for a lack of support for separate compilation with a selective recompilation facility.

Development on Vortex began in the fall of 1993. Initially, Vortex compiled only the Cecil object-oriented language (a pure, dynamically-typed object-oriented language based on multi-methods) [Chambers 93], but it currently is being modified to also compile other object-oriented languages including C++, Modula-3, and Java. Vortex is written entirely in Cecil, and Vortex has been the sole Cecil compiler since the beginning of 1994. All the techniques discussed in this paper are implemented and are used daily by us as part of our ongoing development of the Vortex compiler.

The next section of this paper discusses Vortex's static analyses, including static class analysis and class hierarchy analysis, and considers the impact of whole-program analysis on interactive turnaround time and separate compilation. Section 3 describes using dynamic profiles to guide optimization such as receiver class prediction and considers important properties of profiles, such as peakedness and stability across inputs and program versions, that are necessary for profile-guided optimizations to be effective and practical. Section 4 discusses method specialization, both static and profile-guided. Section 5 reports on our empirical experiments to measure the effectiveness of these techniques on a number of benchmark programs. Section 6 compares the Vortex approach to related work, and Section 7 contains our conclusions and sketches some directions for future work.

2 Static Analyses

One way to reduce the cost of dynamic dispatching is to try to replace a dynamically-dispatched message with a direct procedure call. This is possible, for example, if the receiver argument[*] of the dynamically-dispatched message is known to be an instance of a particular class; in that case, the method lookup can be performed at compile-time and the dynamically-dispatched call replaced with a direct call to the looked-up method. Once the message is statically-bound, then more traditional techniques such as inlining or interprocedural analysis can further optimize the call. Inlining is particularly important as the median size of procedures gets small, as with heavily-object-oriented and functional programming.

[*] For now, we will consider dynamically-dispatched calls where method lookup depends only on the first argument (the receiver), as with Smalltalk and C++. Later we will extend these techniques to consider multi-methods, where method lookup can depend on the run-time classes of any subset of the arguments, as with CLOS and Cecil.

Static class analysis provides a conservative upper bound on the set of classes of which the receiver of a message can be an instance. As with other compiler analyses, class analysis can be performed at a variety of scopes, ranging from local analyses through intraprocedural and interprocedural analyses. Different scopes of analysis offer different trade-offs in terms of precision, implementation complexity, analysis time, and impact on separate compilation. The next section describes intraprocedural class analysis, and section 2.2 describes class hierarchy analysis (a limited form of interprocedural class analysis).

2.1 Intraprocedural Class Analysis

Intraprocedural class analysis computes, at each program point, a mapping from variables to sets of classes. The fundamental safety requirement on these mappings is that, at run-time, the object stored in a particular variable must be a direct instance of one of the classes in the corresponding set. The goal of analysis is to compute the smallest sets possible, since small sets will enable the greatest degree of static binding of dynamically-dispatched messages.

2.1.1 Dataflow Analysis

Intraprocedural class analysis can be structured as a standard forward dataflow analysis. Several representations of class sets are useful:

Name	Definition	Description
Unknown	set of all classes	used if no other information available
Cone(C)	set of all subclasses of C	used for `self` (the receiver) and where static type declarations available
Class(C)	{C}	used for constants, results of `new`
Union(S_1, ..., S_n)	$S_1 \cup ... \cup S_n$	used after control flow merges
Difference(S_1, S_2)	S_1 - S_2	used after unsuccessful type tests

At procedure entry, each formal parameter is bound to Unknown (if in a dynamically-typed language) or to Cone(*C*) (if statically declared to be of type pointer to *C* or one of its subclasses). Relevant flow functions for common operators include:

Analyze([[`x := const`]], ClassInfo_{in}) = ClassInfo_{in}[x → {C_{const}}]*
Analyze([[`x := new C`]], ClassInfo_{in}) = ClassInfo_{in}[x → {C}]
Analyze([[`x := y`]], ClassInfo_{in}) = ClassInfo_{in}[x → ClassInfo_{in}(y)]†
Analyze([[`x := y op z`]], ClassInfo_{in}) = ClassInfo_{in}[x → result(*op*)]‡
Analyze([[`x := msg(rcvr,...)`]], ClassInfo_{in}) = ClassInfo_{in}[x → result(msg)]
Analyze([[`x := obj.var`]], ClassInfo_{in}) = ClassInfo_{in}[x → result(var)]

* C_{const} is the class of `const`.
† `S[x→T]` is the set `S` updated with the binding `x→T`.
‡ `result(x)` represents the set of classes that can result from `x`, whether `x` is an operator, a message, or an instance variable access. For operators, the result classes are pre-defined. For messages and instance variable accesses, the result class set is Unknown (for a dynamically-typed language) or Cone(*C*) (for a statically-typed language where the declared type of the result of the message or the contents of the instance variable is statically declared to be a pointer to *C* or one of its subclasses).

The Vortex compiler incorporates a somewhat more sophisticated analysis that performs a simple kind of alias analysis in parallel with class analysis, enabling the contents of some instance variables to be subject to analysis, for example where a store to an instance variable is followed by a load of the same instance variable.

At merge points, mappings are combined by taking the set union of each of the corresponding class sets for each variable defined in all merging mappings (assuming that all variables are defined along all paths before being used):

$$\text{Meet}(S_1, ..., S_n) = \{v \rightarrow S_{v1} \cup ... \cup S_{vn} \mid v \rightarrow S_{vi} \in S_i, i \in [1..n]\}$$

For most branches, the mappings are propagated to the successor branches unchanged, but for branches that represent run-time class tests, the class set associated with the tested variable is narrowed along the two branches to reflect the outcome of the test:

AnalyzeBranch([[`if x` *op* `y goto L1 else L2`]], ClassInfo_{in}) =
    (L1→ClassInfo_{in}, L2→ClassInfo_{in})
AnalyzeBranch([[`if x in S goto L1 else L2`]], ClassInfo_{in}) =
    (L1→$\text{ClassInfo}_{in}[x \rightarrow \text{ClassInfo}_{in}(x) \cap S]$,
     L2→$\text{ClassInfo}_{in}[x \rightarrow \text{ClassInfo}_{in}(x) - S]$)

Loops can be handled through straightforward iterative techniques. Termination is assured, in the absence of Difference representations of class sets, because the flow functions are monotonic (the size of the class sets at a given program point only grows as iterative analysis proceeds) and there is a finite number of possible class set representations, given the finite number of classes mentioned in the routine being optimized. One can ensure termination in the presence of Difference sets through the use of a widening operator to approximate Difference set representations conservatively when necessary [Cousot & Cousot 77]. If the compiler has access to the entire class hierarchy of the program, as discussed in Section 2.2, this approximation can be avoided by converting the Difference into an exact Union of classes. In our implementation, we also treat Differences whose left-hand operand is Unknown specially, to avoid introducing a recompilation dependency on the set of all concrete classes in the program (as described in section 2.2.4).

2.1.2 Exploiting the Information

The information derived by class analysis is used in two ways: optimizing dynamically-dispatched messages and eliminating run-time class checks. When compiling a dynamically-dispatched send, the class set of the receiver argument is fetched from the mapping before the send. If the class set is bounded (specifies a fixed, enumerable set of classes), then compile-time method lookup is performed for each member of the class set. If all members invoke the same method, then the send is replaced with a direct call to the method (this is more general than static-binding only if the receiver maps to a singleton set). If the classes map to some small number of target methods, then the dynamically-dispatched send can be replaced by a class-case statement which checks the class of the receiver at run-time and then branches to the appropriate statically-bound call. This class-case transformation can improve performance if subsequent optimizations of the statically-bound calls occur. (Class-casing is discussed in more detail in section 2.3.)

Optimizing messages in this way requires that the receiver map to a bounded set of classes. With only intraprocedural information, however, only the Class representation and Unions of Class representations are bounded. In particular, the Cone representation is open-ended, representing an unknown set of classes, since the compiler does not know all the possible subclasses of some class. It is not possible, with only the techniques discussed so far, to perform static binding of messages whose receivers are Cones, since the compiler cannot enumerate all possible receiver classes.

This weakness of Cone representations with intraprocedural class analysis implies that static type declarations of the form that a variable holds a class *C* or any of its subclasses (i.e., those type declarations that support subclass polymorphism, a key component of the flexibility of object-oriented programming) provide no help in optimizing sends through static binding; statically-typed languages are as bad as dynamically-typed languages in this regard. This counter-intuitive result is important, since it suggests that more sophisticated techniques will be needed to optimize heavily-object-oriented programs, whether statically- or dynamically-typed, and that static types alone are insufficient to support optimizing heavily-object-oriented programs well.

Static class analysis also can eliminate run-time class checks; if the result of a class check is implied by the known static information, then the check (and one successor branch) can be eliminated. Class checks may have been introduced as part of run-time checking in dynamically-typed languages (for example, testing whether the argument to + is an integer) or as part of dynamically-checked "narrow" operations as found in statically-typed languages, including the latest version of C++ (dynamic cast), Java (cast), Eiffel (reverse assignment attempt), and Modula-3 (TYPECASE, NARROW).

To get best results, intraprocedural static class analysis should be run in parallel with compile-time method lookup and inlining. This allows the analysis to try to compute a non-trivial class set for the result of the method, based on the class sets of the arguments to the inlined method and the body of the inlined method, to help in improving the quality of analysis downstream of the method. Parallel analysis is particularly important for pure object-oriented languages, where every computation is expressed as a message and hence essentially no useful information would be obtained by an analysis prior to inlining.

2.1.3 Automatic Inlining

The Vortex compiler automatically makes decisions about which routines are appropriate for inlining. Currently, the compiler uses conventional heuristics that examine the apparent size of the callee method, the expected execution frequency of the call (derived from static heuristics [Wall 91] or from dynamic profile data as in section 3), and whether the call is recursive. However, for higher-level languages that use calls for most operations, the heuristics do not perform very well, since they consider only the superficial source-level size of the target method and ignore any effects of optimizations that are performed after inlining. In high-level languages, many of the apparent calls in the target method will turn out after optimization to be simple operations such as arithmetic, standard control structures, or instance variable accesses, leading to a large disparity between the apparent superficial size of a method and the actual size of the optimized code after inlining. To partially overcome this weakness in our heuristics, we allow routines to be annotated with inlining pragmas indicating whether the routine should or should not be inlined, overriding the default heuristics; we use the pragmas on key routines (63 out of 7930 methods in the current implementation of the Vortex compiler itself) to achieve better inlining results.

In previous work, we designed and implemented a more sophisticated automatic inlining decision-maker that overcomes the problems of simple source-level heuristics [Dean & Chambers 94]. When encountering a statically-bound call that might potentially be worthy of inlining, the compiler would tentatively inline the callee and then optimize it in the context of the caller. Only after optimization was completed did the compiler estimate the costs (in compiled code space) and benefits (in instructions removed through optimization) of inlining the routine. If the costs were too high relative to the benefits, then the inlining was undone. To amortize the cost of these "inlining trials" across many call sites that invoke the same method, the compiler recorded the results of its trials in a persistent program database; if cost/benefit information for a particular call was already in the database, then the trial process was skipped and the previous results were used directly. In this way, the compiler trained itself to make inlining decisions based on more accurate cost and benefit information.

An important component of the inlining trial-based system was its treatment of call site-specific information. Optimization of the inlined callee depends on what information is known statically at the call site; different call sites invoking the same routine can lead to quite different effects of optimization. To approximate this dependency of cost/benefit information on the information available at the call site, the compiler would record the static class information known about the call's arguments (since in that system class information about variables is the most important static information). Furthermore, since not all available static information is exploited during optimization, the system recorded which aspect of the available class information was exploited during optimization. These more abstract descriptions of available (and/or lacking) static information were then included with each entry in the inlining database. Only call sites that invoke the same target method with compatible static class information would reuse an inlining trial's database entry. Experimental results showed that this inlining trial-based system led to inlining results that were much less sensitive to superficial changes in the source code's structure and less sensitive to the source-level threshold used to determine whether a routine was potentially a good candidate for inlining.

Our more recent experience with the Vortex source-level heuristics confirms the need for a more sophisticated inlining decision-maker. When examining the results of our compiler, we frequently discover cases where the source-level heuristics are making poor inlining choices, while a decision-maker based on post-inlining costs and benefits would make better choices. We expect to add some version of inlining trials to the Vortex compiler in the near future.

2.1.4 Closure Analyses

Closures (first-class function objects) are used in high-level object-oriented languages like Smalltalk, as well as functional languages like Scheme and ML, for control structures, iterators, predicates, exception handling, and so on. In Smalltalk, closures are used to implement even the most basic of control structures, including `if` and `while`, and closures are heavily-used as part of iterators over collections. However, if unoptimized, heavy use of closures would incur a serious performance cost, in the form of time to allocate (and later garbage-collect) closure objects and the time to invoke the closure's body by sending the closure a message. Consequently, good performance for such languages relies on optimizing away most of the cost of these closures.

Fortunately, techniques for optimizing messages can be applied to optimize closures. By treating each occurrence of a closure expression as a separate class, static class analysis can track the flow of closures through the procedure. Inlining can often optimize away the control structures, leading to code that sends the invoke-closure message (e.g. `value` in Smalltalk) to a variable whose static class is a known closure. This message can itself be inlined, replaced with the body of the closure. If all uses of a closure value are eliminated through inline-expansion, then dead assignment elimination will remove the closure allocation operation. Additionally, if a closure object is used only along some downstream paths (other uses having been inlined away, for instance), the closure allocation operation can be delayed to those program points where it is known that the closure object will be needed at run-time (a kind of partial dead code elimination [Knoop et al. 94]). Using these techniques, Vortex makes most simple uses of closures free, at least in the normal case. Cases that still incur a cost are when a closure is passed as an argument to (or is otherwise visible to) a routine that is not inlined, or when a closure is stored in a data structure that itself is visible to non-inlined routines, but these situations are only a small fraction of the original source-level uses of closures.

2.2 Class Hierarchy Analysis

Intraprocedural class analysis is limited by its lack of information about the program outside of the method being analyzed (and those methods inlined into the method being analyzed). In particular, intraprocedural class analysis has no useful information about the formal parameters of the method or about lexically-enclosing variables, about the contents of instance variables and arrays, or about the results of non-inlined methods. This limitation greatly curtails the effectiveness of strictly intraprocedural class analysis, and overcoming this weakness can lead to overly aggressive inlining in an attempt to improve the quality of analysis.

Interprocedural class analysis could greatly improve the quality of analysis over intraprocedural class analysis, and several research projects have investigated different kinds of interprocedural analyses for object-oriented languages [Palsberg & Schwartzbach 91, Oxhøj et al. 92, Plevyak & Chien 94, Agesen 95, Pande & Ryder 94]. The fundamental complication underlying interprocedural analysis of object-oriented programs is the cyclic dependency between the structure of the call graph and the static class information inferred for message receivers. Early results are encouraging, but some concerns remain about the scalability of these interprocedural analyses to larger programs and higher-level languages [Grove 95] and interprocedural analysis's interactions with separate compilation and fast turnaround time in interactive development environments.

As a simpler alternative to full interprocedural class analysis, the Vortex compiler includes class hierarchy analysis. Class hierarchy analysis augments intraprocedural class analysis with knowledge of the program's class inheritance graph. 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, as well as from static type information in statically-typed languages. In particular, all Cone class sets become bounded, since the compiler can now enumerate all the subclasses of a particular class, enabling them to support compile-time method lookup (as discussed in section 2.1.2). This means that messages sent to the receiver formal of a method as well as all variables with statically-declared types can potentially be optimized, dramatically increasing the opportunities for optimization.

To illustrate, consider the following class hierarchy, where the method *p* in the class *F* contains a send of the *m* message to self:

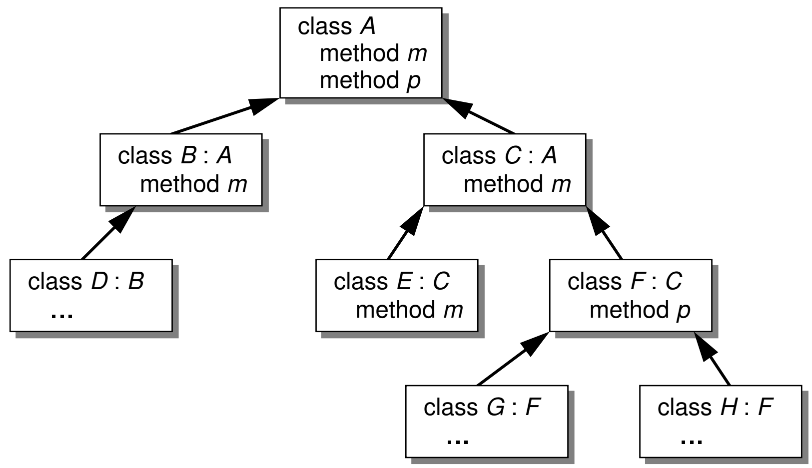


Figure 1: Class Hierarchy Example

In general, sends of the *m* message must be dynamically-dispatched, since there are several implementations of *m* for subclasses of *A* (in C++ terminology, *m* is declared as a virtual function). 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, traditional 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*.

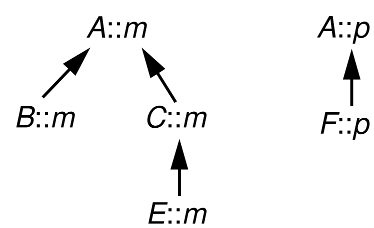
A key aspect of class hierarchy analysis is that it supports the goal that the program will only incur the cost of dynamic dispatching where it is used in the program. If a message invokes a particular method, and the method is known not to be overridden, class hierarchy analysis will enable the message to be replaced with a statically-bound call. This facility supports the development of flexible, reusable class libraries and frameworks: the libraries can be constructed using solely dynamic dispatches for interaction, supporting a high degree of *potential* extensibility, but only the extensibility used by a particular application *in practice* will incur the cost of dynamic dispatching. Unused potential will be optimized automatically and simply into direct procedure calls open to inlining.

2.2.1 Implementation of Compile-Time Method Lookup

Most previous implementations of compile-time method lookup considered only receivers that were a particular class. With Cone and Union class sets, however, the receiver of a message can be some bounded enumerable set of classes. As long as all members of this set inherit the same method, static binding is still available. Non-singleton receiver class sets are important: studies of compiling the Vortex compiler itself indicate that nearly 50% of the messages statically-bound using class hierarchy analysis have receiver class sets containing more than a single class.

One implementation of compile-time method lookup in the presence of non-singleton receiver class sets 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 developed an alternative approach for Vortex 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 (i.e., those classes that inherit from the class where the method is defined and do not inherit some overriding method); we call this set the *applies-to* set. (Vortex computes 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 checking a large number of methods for applicability at every call site in the program, Vortex 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 earlier example, we would construct the following partial order:



Then for each method, we initialize its applies-to set to Cone(C), where C is the class where the method is defined. 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 $O(N^2)$,* 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(C_1) overlaps Class(C_2) iff $C_1 = C_2$ or C_2 inherits from C_1. 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

* 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).

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 bit-set representation. Overlaps testing of two bit 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 bit-set 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.

2.2.2 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 checks need be made during compile-time method lookup 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 treat 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 (due to multiple inheritance, for instance), more than one method will include the class in its applies-to set, again preventing static binding to either method.

2.2.3 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.* 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.

* 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 upon which any of the methods is specialized. 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.

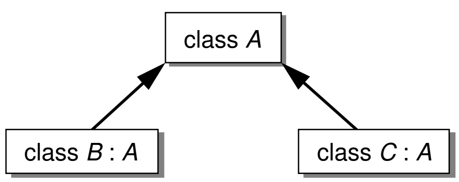
Static class analysis is modified to support multi-methods as follows. For each method, we precompute the method's applies-to *tuple* of class sets; this tuple describes the combinations of classes for which the method should be invoked. For a multi-method specialized on the classes C_1, ..., C_k, the method's applies-to tuple is initialized to $\langle \mathrm{Cone}(C_1), ..., \mathrm{Cone}(C_k) \rangle$. 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, is incorrect. One straightforward and correct representation would be a union of *k* *k*-tuples, where each tuple has one element class set difference taken:

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

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.

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:

```
m(@A, @A, @A)
      ↑
m(@B, @C, @A)
```

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*): $<\{A\}, \{A,B,C\}, \{A,B,C\}> \cup <\{A,C\}, \{A,C\}, \{A,B,C\}> \cup$
$<\{A,C\}, \{A,B,C\}, \{A,B\}> \cup <\{A,B\}, \{A,B\}, \{A,B,C\}> \cup$
$<\{A,B,C\}, \{A\}, \{A,B,C\}> \cup <\{A,B,C\}, \{A,B\}, \{A,B\}>$

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 reduces to the following:

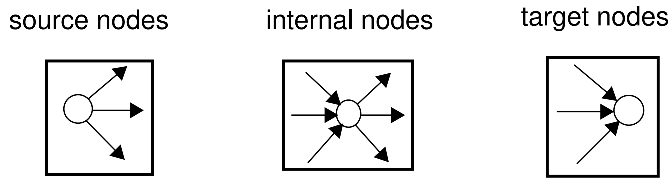
m(@*A*, @*A*, @*A*): $<\{A\}, \{A,B,C\}, \{A,B,C\}> \cup <\{C\}, \{A,C\}, \{A,B,C\}> \cup$
$<\{C\}, \{B\}, \{A,B\}> \cup <\{B\}, \{A,B\}, \{A,B,C\}>$

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 (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.

2.2.4 Incremental Programming Changes

Class hierarchy analysis, as a kind of interprocedural analysis, might seem to be in conflict with incremental compilation: the compiler generates code based on 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.

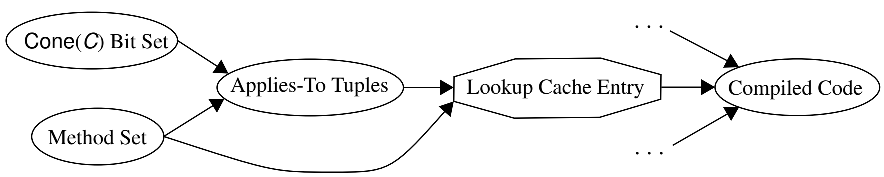
Fortunately, class hierarchy analysis can be applied even during active program development if the compiler maintains enough intermodule dependency information to be able to selectively recompile those parts of a program invalidated after some change to the class hierarchy or the set of methods. The Vortex compiler includes such a framework for maintaining intermodule dependency information. This subsection serves as an overview of the dependency mechanism; a more detailed description is available elsewhere [Chambers et al. 95]. 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.

To illustrate how this dependency framework is used in Vortex, 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 bit-set 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 dependency node represents the bit-set representation of the set of subclasses of a class (one product of class hierarchy analysis),
- another dependency node represents the set of methods with a particular name (another product of class hierarchy analysis),
- a third dependency node represents the applies-to tuples of the methods, which is derived from the previous two pieces of information, and

- a fourth dependency node guards each entry in the compile-time method lookup cache, which was computing from the set of methods and their applies-to tuples.

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 not invalidating downstream nodes 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; further details of Self's mechanism were published in [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 kinds of programming changes, and even if few compiled files become invalidated, a fair amount of compilation time is expended in checking caches. Also, 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.

2.2.5 Separate Compilation

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 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. This approach will not optimize the library code for the enclosing application, however, losing some of the benefits of class hierarchy analysis.

To get the full benefits of class hierarchy analysis while still supporting separate development of libraries and protection of proprietary source code, a library could be delivered not as binary code but instead as intermediate code. With intermediate code, the compiler can optimize the library for the specific way it is used and extended by the application, eliminating unneeded generality in the library code. As long as it is sufficiently difficult to recover the library's source code from the intermediate code, the source code will remain protected. The approach of compiling to intermediate code rather than binary code is also taken by Fernandez [Fernandez 95], but to support link-time optimization rather than separate compilation of libraries.

Even though compilation may not be separate, static reasoning, typechecking, and verification can be done separately. We believe strongly in the ability to reason about modules in isolation from their prospective clients, and vice versa. The use of whole-program analysis by the compiler to aid in generating high-quality code does not affect the localized program view used by humans and typecheckers during static reasoning.

2.2.6 Annotations versus Class Hierarchy Analysis

Some object-oriented languages allow the programmer to provide annotations that give some of the benefits of class hierarchy analysis. 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,* 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 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

* 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.

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 particular subtree of the inheritance graph, there will be only one implementation that applies. In the hierarchy previously depicted in Figure 1, 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 5, 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, Dylan [Dyl92] allows a class to be sealed (indicating that it will have no subclasses other than those statically-visible in the same compilation unit), and Java provides the most general mechanism by allowing both classes and methods to be declared final [Gosling et al. 96] (indicating that the class or method won't be subclassed or overridden); a final method can override some (non-final) method, overcoming the limitation of non-virtual functions described in the third bullet above. Class hierarchy analysis automatically determines the same or better information than these approaches, however, and it adapts to the actual usage of a class library by a particular program.

2.3 Exhaustive Class Testing

Class hierarchy analysis may not be 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. The histogram in Figure 2 shows the distribution of the number of potentially invoked methods that were identified by class hierarchy analysis at the 52,891 message send sites that were analyzed during the compilation of 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.

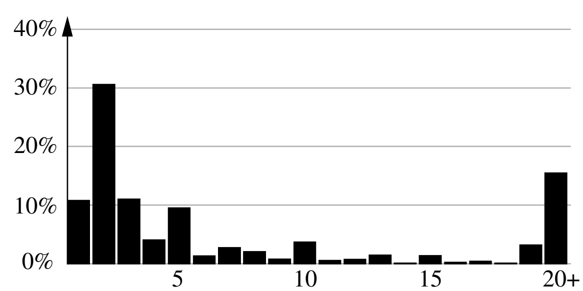
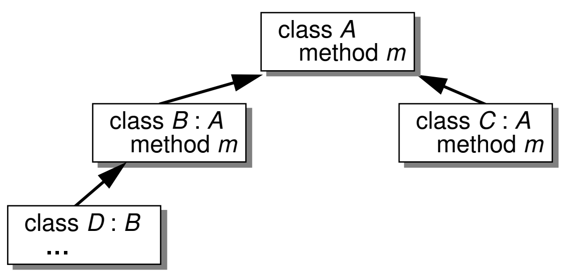


Figure 2: Number of candidate methods per static call site

This histogram shows that only 10% of the static call sites had only a single candidate method, but that 55% of the call sites had between 2 and 5 candidate methods. In this section, we explore the possibility of inserting explicit tests to partition all the potential classes at a call site among the various methods that might be invoked. This exhaustive class testing produces code in which a message send is converted to several statically-bound calls, providing two main benefits. First, all paths are statically-bound to a single routine, and, depending on how dynamic dispatches are implemented, the tests that are inserted to effect this static-binding can be faster than a full dynamic dispatch. Second, if all of these statically-bound calls are inlined, there are often downstream benefits from knowing more precise information about the result of the message send. Section 2.3.1 and Section 2.3.2 consider two mechanisms for supporting class tests, and Section 2.3.3 describes an algorithm that relies on these mechanisms to insert exhaustive tests to statically-bind a send.

2.3.1 Single Class Tests

All object-oriented language implementations are able to compute some unique identifier of the class of an object 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, and a map identifier in Self and Cecil all serve the purpose of uniquely identifying an object's class. Using the class identifier, a test to see if an object is a member of a particular class can typically be implemented in only a few instructions on most modern machines, as a comparison of an object's class identifier against a compile-time constant representing the class. The low cost of such tests 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. Class hierarchy analysis will be able to narrow down the possible methods invoked




```
A* obj = ...;
x = obj->m();
```

Figure 3: Example class hierarchy and code for class tests

by the send obj->m() 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 effect, the compiler translates a message send into an equivalent sequences of inline code that is similar to a typecase statement, as in Modula-3 [Nelson 91] or Java [Gosling et al. 96]. One possible sequence of tests is shown in Figure 3.

Unfortunately, there are two aspects of individual class tests that limit their general utility as an exhaustive class testing mechanism:

- *Testing for large hierarchies is impractical*. A second problem, limiting the cases in which exhaustive class testing can be applied, is that testing for singleton classes becomes impractical when a large number of classes all invoke the same method, 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 class testing performed in Figure 3 would be impractical because it would require too many tests, causing a large blowup in compiled code space and, assuming that the compiler had no

```
A* obj = ...;                 A* obj = ...;
x = obj->m();                 id = obj->class_id;
                         ⟹    if (id == D or id == B) then
                                 x = obj->B::m();
                              else if (id == C) then
                                 x = obj->C::m();
                              else
                                 x = obj->A::m();
```

Figure 4: Static binding through exhaustive tests

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*. Exhaustive class testing using singleton class tests for specific classes 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 class testing done in Figure 3 would no longer be correct for the call site and it would need to be recompiled.

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

2.3.2 Cone Tests

If a language's implementation is able to provide a means of quickly testing subclassing relationships, then both of the drawbacks discussed in Section 2.3.1 can be mitigated, and exhaustive class testing can be practically applied to a much larger fraction of call sites. For example, the compiler would often like to be able to generate tests of the form:

```
if (obj.class_id inheritsFrom class B) then ...
```

Such tests can be termed "cone tests" because they test whether a particular object's class lies in the cone of all classes inheriting from a particular class. The way in which such a test is implemented 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 has y as an ancestor" **iff** $x.l \geq y.l$ and $x.h \leq y.h$

A simple algorithm to assign l and h numbers to each class is to traverse the class hierarchy tree(s), assigning l values in a preorder walk, and h numbers in a postorder manner. Once the class hierarchy has been assigned these numbers, then a subclass test of the form "`A inheritsFrom B`" can be implemented as a sequence of two loads to extract the class identifier plus a pair of comparisons for each subclass test. This approach was inspired by the DEC SRC Modula-3 compiler, which implements user code containing `typecase` statements in a similar manner [SRC].

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 computes an $N \times N$ boolean matrix of subclass relations, 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. 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 matrix is represented as a bit matrix, the code in Figure 5 can be used to test the inheritance relationship.

```
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 5: **inheritsFrom(obj,B)** implementation with multiple inheritance

Encoding inheritance relationships in this manner allows an `inheritsFrom` relationship to be tested using a six instruction sequence: two or three `load` operations, followed by an `and`, `cmp` and a `beq` on most modern processors (although there is no instruction-level parallelism among the instructions, since each instruction is dependent on the result of its predecessor).

By encoding the matrix to use only a single bit per entry, this approach consumes N^2 bits of space. 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. Alternative encodings of the `inheritsFrom` relationship are possible, representing other time-space tradeoffs: Aït-Kaci *et al.* provide a useful overview of efficient lattice operations that discusses many of these alternatives [AK et al. 89].

Table 1 presents a summary of the pros and cons of single class tests and cone tests.

Table 1: Class testing alternatives

Kind of test	Instruction Cost	Cons
Singleton class test	5 (1 load, 3 ALU, 1 branch)	Only tests single class
Cone test (single inheritance)	6 (2 loads, 2 ALU, 2 branches)	Only applicable for single inheritance
Cone test (multiple inheritance)	5-6 (2-3 loads, 2 ALU, 1 branch)	Space: Subclass relationship matrix requires $O(N^2)$ space

2.3.3 Exhaustive Class 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 unreasonably. 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` cone test described in Section 2.3.2.

The input to the exhaustive class 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:

- *Small number of 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.
- *All 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, because of downstream benefits. Our current heuristics only consider exhaustive class testing when all *N* candidate methods are inlinable.

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 (as defined in Section 2.2.1, repeatedly removing one of the bottom (most-specific) methods of the partial order and generating exhaustive tests for this method, until the partial order is empty. The algorithm is shown in Figure 5.

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 this method or some overriding 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* in candidate methods **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 path of successful tests, 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 6: Algorithm to perform exhaustive class testing

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 for, 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 first test for methods that are interior nodes in the partial order, 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. In practice, the performance improvements possible by testing for an interior method first are limited, since we only consider applying exhaustive class testing when the number of candidate methods is small.

2.4 Additional Static Analyses

2.4.1 Value-Based Analysis

In the presence of heavy inlining, the intermediate representation of procedures becomes littered with simple `x := y` copy instructions, introduced to copy actuals to formals at entry to the callee and similarly for the result at exit from the callee. These copy statements lead to a situation where multiple variable names all hold the same run-time value at a given program point. In the presence of intermediate-language instructions that modify the information associated with a variable, such as a run-time class test instruction which narrows the static class information associated with its argument variable along its two downstream paths, the information associated with the copies may not be updated accordingly.

To ensure that information associated with one name is propagated to all "equivalent" names, the Vortex compiler follows the Self compiler [Chambers 92] in introducing value intermediate objects. During dataflow analysis, each defined variable name is mapped to a value object, which represents the possible run-time values that can be stored in that variable at run-time. The following kinds of values are supported:

Name	Description
UnknownValue(*id*)	unknown value: only known to be the same as UnknownValues with same *id*; represents the values of incoming arguments, results of non-inlined calls, and contents of otherwise unknown instance variables and array elements; at merges of different values, a new UnknownValue is created to represent the union, in a manner similar to ϕ functions in static single assignment form [Cytron et al. 89].
ConstantValue(*value*)	the constant *value*
UnopValue(*op*, *value*)	the result of a unary operator that was not constant-folded
BinopValue(*op*, $value_1$, $value_2$)	the result of a binary operator that was not constant-folded

Multiple variable names can map to the same value object. In particular, a copy statement simply updates the left-hand-side variable to map to the same value as does the right-hand-side variable. Dataflow information is then associated with values instead of variables. In Vortex, this includes static class information, available expressions for common subexpression elimination, and so on. Constant propagation and copy propagation become easy to express as replacing a variable reference with a canonical value or variable name based on the value associated with the name.

It might be possible to get an effect similar to Vortex's value-based analysis by performing copy propagation before analysis (particularly if converting to static single assignment form [Cytron et al. 89] before copy-propagating). However, since Vortex makes fairly sweeping changes to the control flow graph as part of optimizations such as inlining and splitting (described later in section 3.2.2), introducing new copy statements and duplicating others, maintaining static single assignment properties and propagating the new copies could become expensive and unwieldy. The explicit variable→value mapping can be computed in parallel with other dataflow information fairly easily.

2.4.2 Instance Variable Optimizations

Heavily-object-oriented programs tend to allocate objects fairly frequently. These objects are often initialized by separate constructor methods, perhaps modified after allocation by the caller, and often manipulated for a short time and then thrown away. In the presence of inlining, the object creation, initialization, modification, reading, and deallocation may all occur within a single compiled procedure.

Vortex includes some simple optimizations that strive to reduce the number of object instance variable writes and reads and potentially even eliminating intermediate objects altogether. In parallel with computing available expressions for common subexpression elimination, Vortex tracks the contents of memory locations, in the form of a set of (*base*+*offset*)→*contents* bindings, where *base*, *offset*, and *contents* are all values. Two *base*+*offset* addresses are the same only if the *base* and *offset* components are known to be the same. If derived pointers are not allowed in the intermediate language, two *base*+*offset* address are known to be different if either *base* or *offset* components are known to be different; if derived pointers are allowed, then a more conservative rule is used.

When analyzing a store instruction of the form `*(base+offset) := contents`, where the operands of this instruction map to the values *base*, *offset*, and *contents*, the memory map is consulted to see if there exists a binding of the form (*base*+*offset*)→*contents'*. If found, then *contents* and *contents'* are compared. If they are known to be the same, then the store instruction is redundant and eliminated, otherwise the old binding is replaced with one mapping to *contents* (a "strong update"). If no definite match is found, then a new binding (*base*+*offset*)→*contents* is added to the map (a "weak update"). Several bindings may exist for memory locations that may or may not be the same, representing a "may-alias" relationship.

When analyzing a load instruction of the form `result := *(base+offset)`, where the operands of this instruction map to the values *base* and *offset*, the memory map is consulted to see if there exists a binding of the form (*base*+*offset*)→*contents*, i.e. a "must" binding for this memory location. If there is, then this load is redundant and is replaced with `result := contents`, where `contents` is any variable holding *contents* at that program point (the necessary mapping from values to distinguished representative variables is already supported to implement copy propagation). By performing redundant load elimination in parallel with static class analysis, any class information known about a value before it is stored into an instance variable can be recovered when it is subsequently loaded from the instance variable.

In addition to identifying these redundant load and store instructions, Vortex identifies when a store is dead, i.e., when its contents will never be read. Dead stores sometimes arise when an (inlined) object constructor initializes instance variables which are then subsequently reassigned to more appropriate values by the caller. In a manner similar to dead assignment elimination, Vortex performs a reverse dataflow analysis tracking which memory locations are live. At the return of the procedure, all memory locations are considered live. (This will be refined below.) Before a store instruction, the target memory location is dead (since it will be overwritten). Before a load, the target memory location is considered live. Before calls, all memory locations are considered live. A store is dead if at the point after the store the target memory location is known to be dead. Dead stores are eliminated.

Some objects are allocated by a procedure and never escape to the outside world (e.g. by being passed as a parameter to a call, assigned to an (escaped) object's instance variable, stored in a global variable, or returned from the procedure). For these local objects, a more aggressive dead store identification rule can be used. At the return of the procedure, all instance variables of local objects are considered dead rather than alive, and calls do not affect the state of instance variables of local objects. A store whose *base* pointer is a local object is dead either if there was a later store to a memory location known to be the same or if there were no later loads of memory locations that may have been the same.

Once all redundant and dead loads and stores have been removed, the only remaining reference to an object may be its allocation. In this case, the object itself is dead. Dead assignment elimination takes care of deleting the allocation instruction.

3 Profile-Guided Optimizations

Static analyses work well when they can identify that only one or two cases are possible. However, if many cases are possible, then static analysis provides little benefit. To supplement static analysis, dynamic profile data can be used to predict which of the many possible cases are most common, allowing the compiler to try to optimize for the common case. Vortex exploits several different kinds of dynamic profile data to improve on purely static analysis:

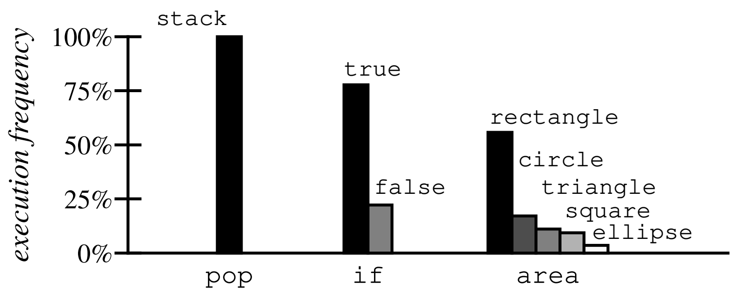
- Vortex uses dynamic execution frequency data to assist in making inlining decisions and to scale down optimization for infrequently-executed routines.
- Vortex uses dynamic receiver class distribution data gathered for messages to guide receiver class prediction, an optimization where run-time class tests are inserted before the send to handle the commonly-occurring receiver classes.
- Vortex also uses dynamic receiver class distributions to guide selectively producing specialized versions of methods for commonly-occurring receiver (and argument) classes.

The first point above is similar to how other compilers have used profile data [Chang et al. 92]. The rest of this section describes the last two points which are more specific to object-oriented languages. The next subsection discusses the granularity of receiver class distribution data. Subsection 3.2 describes profile-guided receiver class prediction. Subsection 3.3 discusses some statistical properties of profile data such as peakedness and stability which are important for profiles to be useful, accurate predictors of future execution behavior. Subsection 3.4 concludes with a description of how we gather profile data efficiently.

3.1 Receiver Class Distributions

The key new kind of profile information available in object-oriented languages is an execution frequency histogram for each of the classes of the receiver of a message. These receiver class distributions indicate which classes are most common for a given message, and moreover they indicate whether one or a few classes dominate or whether the distribution is fairly flat. In a system with multi-methods, each element of the distribution can be a n-tuple of argument classes, where n is the number of specialized arguments for the message; to simplify terms, we will use the term "receiver class" to refer either to a single class (in a singly-dispatched language) or to one of these n-tuples (for a multiply-dispatched language).

The following diagrams illustrate some sample receiver class distributions:



The `pop` message has only a single receiver class, `stack`, in practice. The `if` message has a `true` receiver three quarters of the time; this receiver class information supports a kind of branch prediction. The `area` message is sent to many receiver classes, but `rectangle` instances are by far the most common.

3.1.1 Context for Receiver Class Distributions

A receiver class distribution is associated with some set of messages. The size of this set reflects a trade-off between precision of the information (for small sets) and generality of the information (for large sets). At one extreme, a distribution could be associated with all messages of a particular name, producing a *message summary* distribution. If call sites sending a particular message have similar distributions, then a single message summary distribution is an adequate predictor. However, it is often the case that different call sites of a particular message have different distributions. A more precise approach would associate receiver class distributions with a particular call site, producing *call-site-specific* distributions, to preserve variations across call sites and avoid "smearing together" different profiles. It is possible to exploit even more context, associating a receiver class distribution with a particular stack of calling methods enclosing the call site, up to some height *k*. Such *call-chain-specific* distributions avoid blending together distributions from different clients of a method, such as with shared polymorphic library routines that are used differently by different parts of a program. In general, it probably would be more expensive to gather call-chain-specific distributions for arbitrary *k*, but in the presence of inlining, the call chain whose height is the number of levels of inlined callers can be gathered with no more cost than a call-site-specific distribution.

We unify these different degrees of context for profile information under the *k*-CCP (Call Chain Profile) model [Grove et al. 95], patterned after Shivers's *k*-CFA family of control flow analyses for Scheme [Shivers 88, Shivers 91]. A *k*-CCP receiver class distribution is associated with the message being sent and the specific call sites within the *k* procedures that dynamically enclose the call. A 0-CCP distribution is associated only with the message being sent, leading to a message summary distribution. A 1-CCP distribution is specific to a particular call site within a particular procedure, producing a call-site-specific distribution. Higher values of *k* reflect the additional context of a chain of *k* enclosing procedures. We use *n*-CCP to refer to distributions that use the maximum amount of inlined call chain context available; *n* will stand for different *k*'s for different call sites. This same framework can be used to model the context associated with other kinds of information about calls, such as the execution frequency of a call in different contexts.

As the amount of context increases, the precision of the distributions increases, but the applicability of the distributions decreases; fewer call sites of a program being compiled will be able to use a receiver class distribution as *k* increases. Information with less context can be used more widely; in particular, message

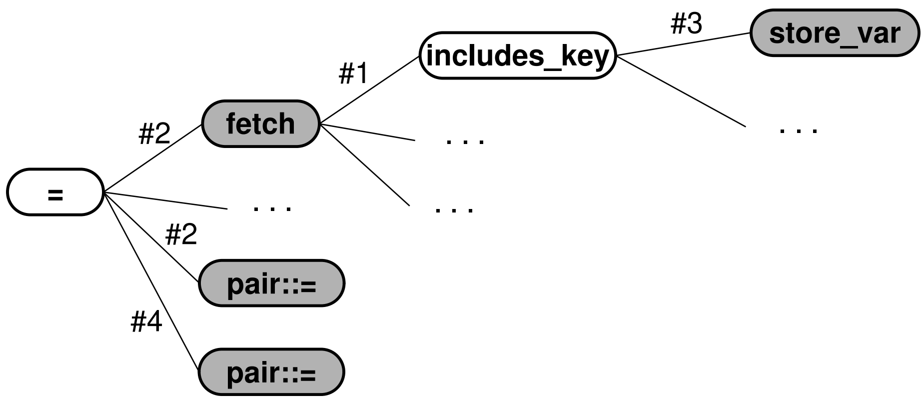


Figure 7: Call-Chain Representation

summary (0-CCP) distributions can be used to predict the behavior of call sites that have not been profiled at all. Fortunately, it is relatively straightforward to combine multiple high-context distributions to calculate the corresponding lower-context distribution. Consequently, the Vortex compiler maintains receiver class distributions using the greatest context available, but summarizes distributions as necessary to construct distributions for call sites with less available context.

3.1.2 Representing and Summarizing Receiver Class Distributions

In a given profile, there may be many different distributions associated with a given message, for different partially-overlapping call chains of varying lengths. Internally, we represent the profile information for a given message as a tree, where each node represents a particular call chain of length ≥ 0 and common prefixes of call chains have been factored. Each node may or may not have a receiver class distribution associated with it; all leaves have information. For example, Figure 7 shows four receiver class distributions for the = message (shaded nodes have distribution information). The numbers along the arcs indicate the call site within the enclosing procedure. Two separate call-site-specific distributions are being recorded for the two distinct sends of = in the pair = method, a distribution is being recorded for the = send within the `fetch` method shared by all table-like data structures, and a separate distribution is being maintained for the same = send in the `fetch` method when it is inlined within the `includes_key` method that is itself inlined within the `store_var` client method. It is likely that this latter dynamic occurrence of the = message will have a quite different (and more precise) receiver class distribution than the general distribution for = within `fetch`.

During compilation, to extract the distribution for a particular message send, the compiler searches the message's prediction tree, identifying the longest call chain prefix common to the call site and the tree. If that node has distribution information, the compiler uses it directly, assuming it to be the most accurate predictor of future call sites with that chain. If the node does not have profile data, it must have successor nodes (i.e., nodes indexed by longer call chains) with data, and a summary distribution for that node is calculated by summing the counts of each class of the successor distributions. For example, if a call site sending the = message is being compiled, but there is no profile data yet for that call site, then its call chain will share only the first node with the profile database's call tree for =. Since that node has no profile data of its own, all its successor nodes will be aggregated into one summary distribution for =, and cached in that node for future reference. In this fashion, global message summaries are calculated on demand from the more specific distribution information. Similarly, summaries at finer granularities can be calculated when needed.

3.2 Receiver 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 call 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 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 7, tests for the two most common receiver classes at some call site have been inserted:

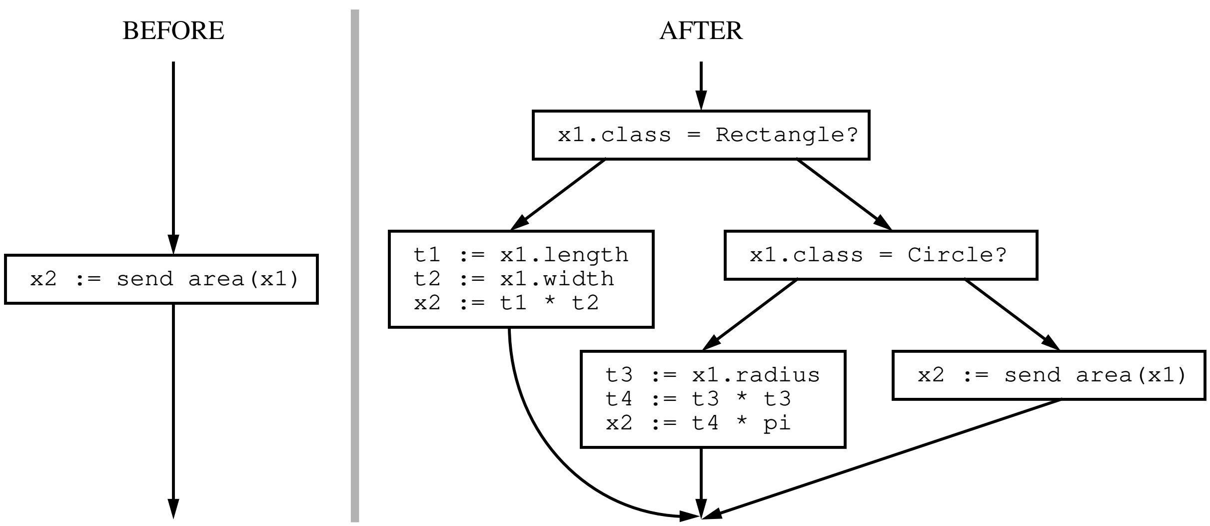


Figure 8: Control flow graph before and after insertion of class prediction tests

The predicted classes can speed up as long as the benefits of avoiding the dynamic dispatch or of optimizing the callee in the context of the caller is greater than the run-time cost of a class test. 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 unpredicted cases.

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.

3.2.1 Sources of Receiver Class Distributions

Earlier compilers for dynamically-typed 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 + was expected to be sent to `SmallInt` instances and `ifTrue:ifFalse:` to `True` or `False` instances. In effect, this hard-wired table provided O-CCP (message summary) receiver class distributions for a small set of basic messages.

Compilers for other dynamically-typed languages like Scheme often incorporate similar optimizations to speed the performance of generic arithmetic.

Receiver class distributions derived from dynamic profile information can greatly improve upon hard-wired receiver class distributions. First, they more accurately reflect actual program behavior rather than estimates built into the compiler before the program was written. Second, all messages in the program can have distribution information, not 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 (1-CCP) and even call-chain-specific (n-CCP) monitoring of receiver classes to be more precise than simple message summary (0-CCP) 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, n-CCP distributions can flag those + messages that are likely to have `SmallFloat` arguments.

The Vortex compiler supports both hard-wired and profile-derived distributions. Upon start-up, a standard file is read in that "seeds" the receiver class distributions for standard messages. These distributions are used if no profile-derived data is available. If profile-derived distributions are gathered, they can be read into the Vortex compiler, replacing the earlier default distributions. We have not investigated how to combine multiple profiles into a single profile, but this would be a useful idea to explore.

Vortex assumes that profiles are gathered off-line in a separate training run of the application, and then the program is compiled statically using these previously-gathered profiles. This approach works well if 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, as discussed more in section 3.3, but this may not be true of some programs. The latest Self compiler [Hölzle & Ungar 94a] avoids this issue by exploiting 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 it 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 (including extra run-time overhead for compilation, extra run-time memory overhead for the compiler, its internal data structures, and a representation of the program being dynamically compiled, and the inability to easily share dynamically-compiled code across multiple users) as well as the extra run-time costs of gathering receiver class distributions for all program executions.

3.2.2 Splitting

Receiver 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 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 7 [Chambers & Ungar 90].

Vortex performs splitting in a lazy fashion. During forward static class analysis, the compiler tracks which variables' class sets have been "diluted" at merges, i.e., where the set of possible classes of a variable is

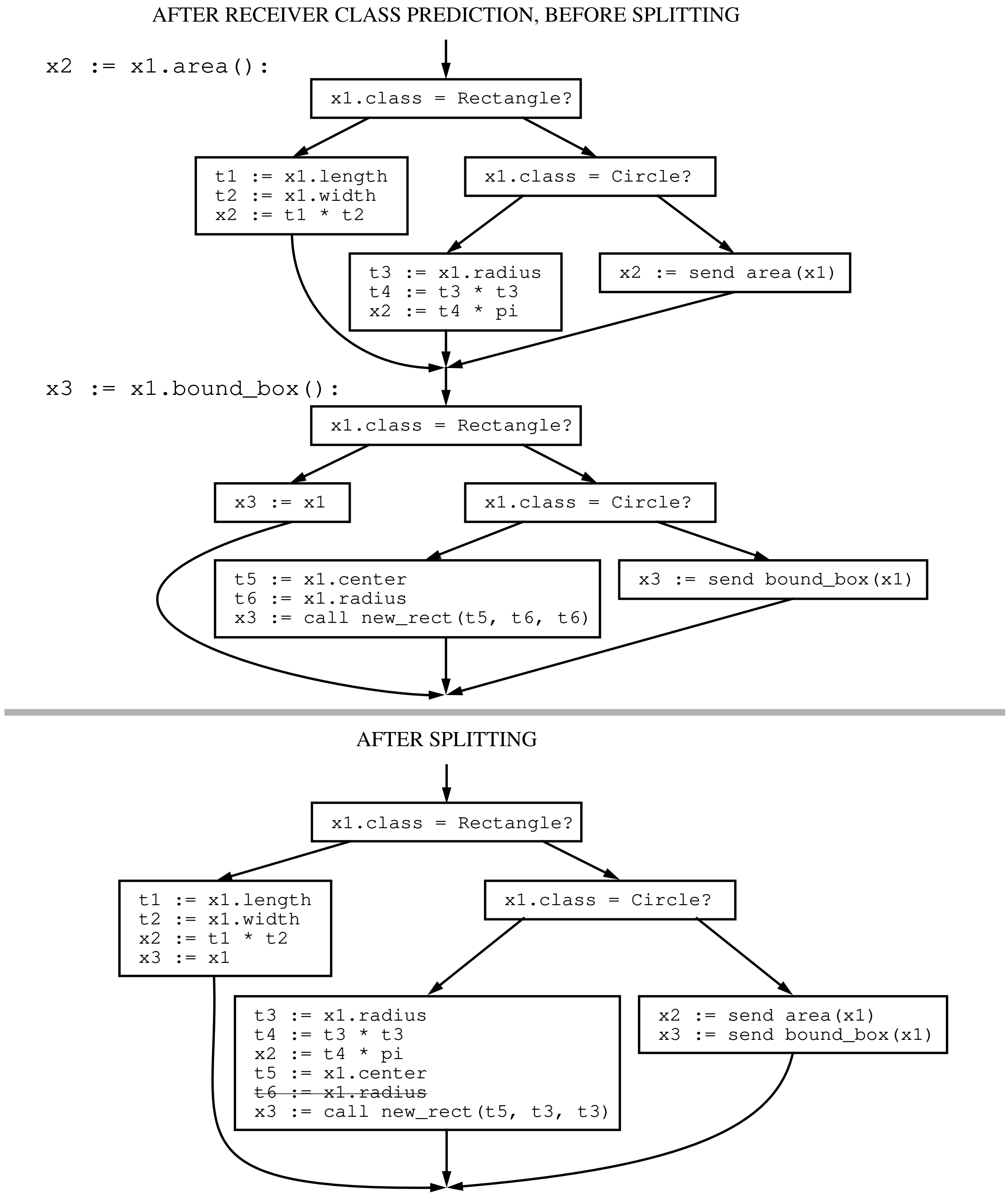


Figure 9: Control flow graph before and after splitting to eliminate redundant tests

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 type 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 accumulated 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].

The current Vortex compiler does not support splitting past a loop entry or exit node, to simplify the analysis. 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 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 7.

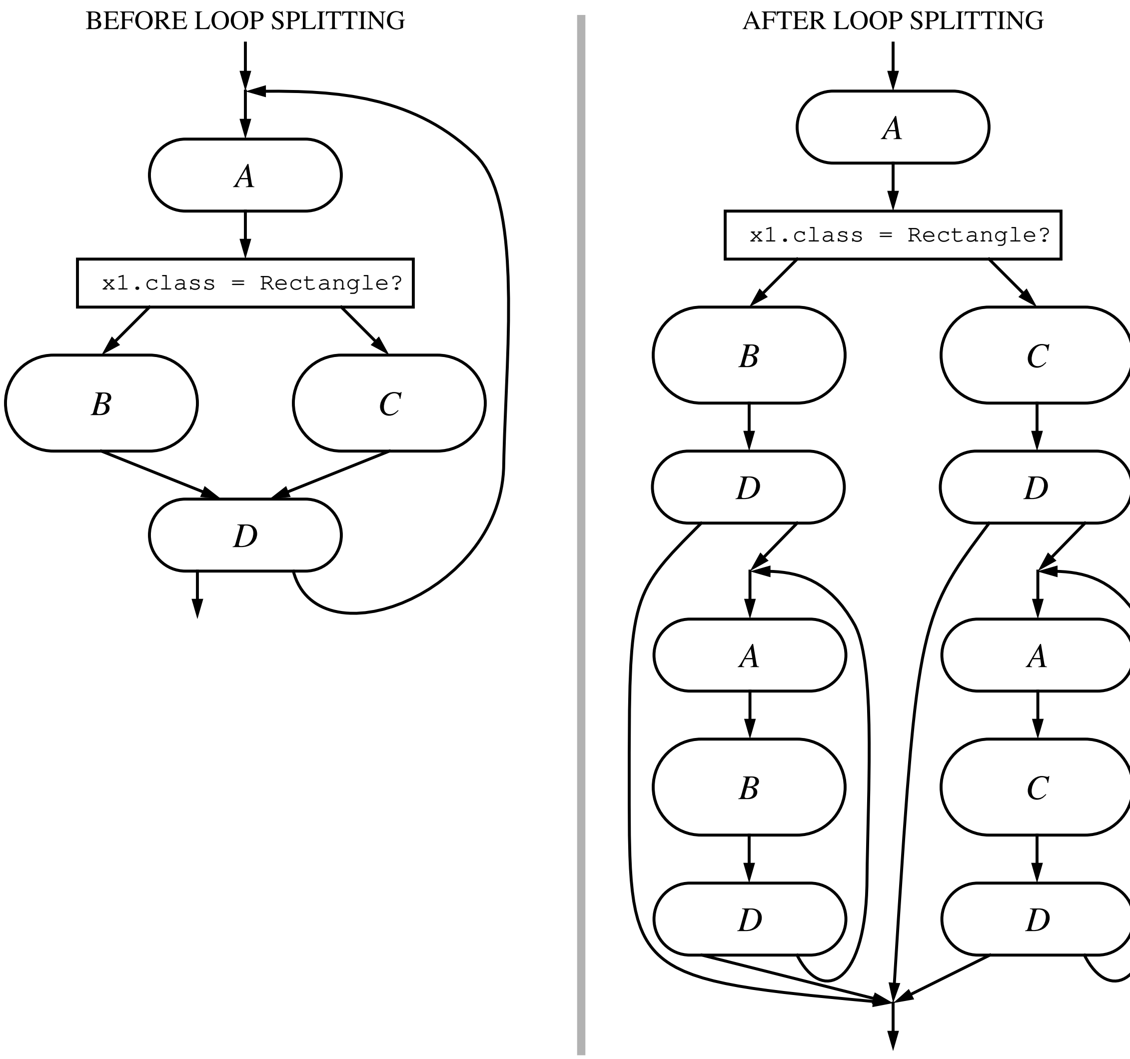


Figure 10: Loop splitting to create multiple loop bodies

Since splitting in general and loop splitting in particular can increase compiled code space substantially, the Self compilers exploited dynamic compilation by deferring compilation of paths downstream of uncommon branch outcomes [Chambers & Ungar 91]. For example, if an operand of a + message is expected to be an integer virtually all the time, then a run-time class test is compiled into the program, but the test-failed branch leads to a stub. If ever taken at run-time, the stub will invoke the compiler to compile the uncommon path. Vortex, being a static compiler, does not exploit lazy compilation of uncommon branches.

3.3 Qualities of Profiles

For profile-guided receiver class prediction to have an opportunity to improve upon the results of static analysis, the receiver class distributions should be strongly peaked, i.e., some small number of receiver classes should dominate execution and so be worth predicting for. In addition, for off-line profiling to be effective, profiles from one program execution (perhaps a special training execution) should be predictive of the behavior of future executions, i.e., profiles should be stable across program inputs. Moreover, in an interactive development environment, we would like to amortize the effort to gather a profile over multiple versions of the program under development, and so we would like the profile of one version of a program to predict the behavior of future versions, i.e., profiles should be stable across program versions.

To assess the degree to which profile-derived receiver class distributions are peaked and stable, we gathered profiles from several reasonably large C++ and Cecil programs that we deemed were written in a fairly heavily-object-oriented style:

Language	Program	Size (in lines)	Inputs
C++	Self-91 compiler	33,500	Small Self benchmarks, Cecil compiler written in Self
	Self-93 compiler	14,900	Small Self benchmarks, Cecil compiler written in Self
	`doc` editor	15,400 + 24,900 library	Typing in one page of text. Randomly cutting and pasting in an existing 10 page document.
	`idraw` graphical editor	6,300 + 24,900 library	Drawing lots of rectangles. Exercising all possible shapes and text.
	Trace-driven memory subsystem simulator	22,000	Memory traces from the execution of the `gcc` and `doduc` programs
	teaching compiler for undergraduate course	4,600	Several small programs
Cecil	Global instruction scheduler for the MIPS	2,400 + 7,400 library	Several MIPS assembly files
	Vortex compiler	38,000 + 7,400 library	Towers of Hanoi benchmark and compiler test suite, with and without optimization

All of the applications were compiled using standard intraprocedural optimizations. For the C++ programs, we used `g++` version 2.3.3 with an optimization level of `-O2`. Because we are interested only in call sites that would be potential candidates for profile-guided class prediction, we instrumented virtual function calls only. We did not measure the class distributions for non-virtual member functions, since these call sites are already statically bound and consequently have no need for class prediction. The Cecil programs were compiled using hard-wired class prediction for common messages, intraprocedural static class analysis, splitting, inlining, and standard intraprocedural optimizations. This default level of optimization was chosen

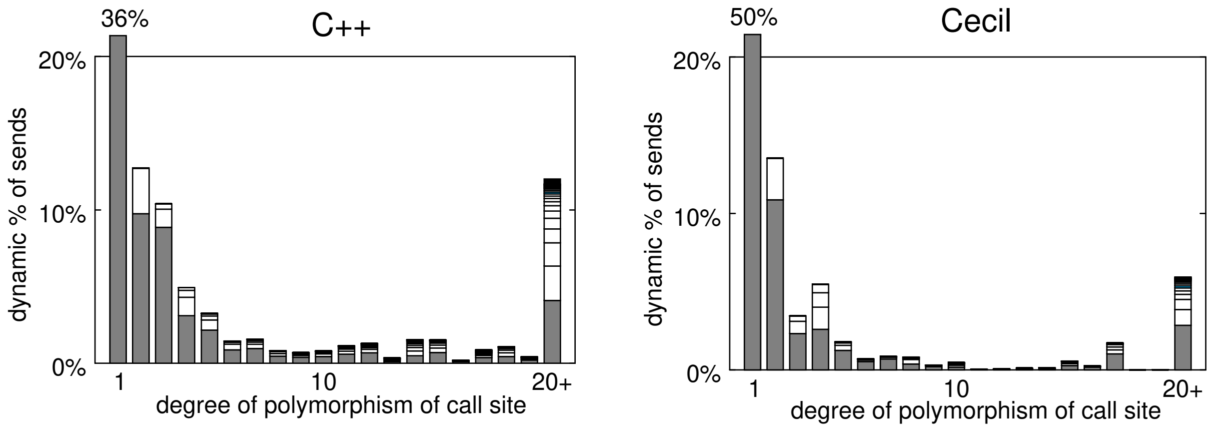


Figure 11: Receiver Class Polymorphism

to compensate for the pure object-oriented model and user-defined control structures in Cecil, eliminating dynamic dispatches due to simple arithmetic operations like + and control structures like `if` that are not present in languages like C++ which have a selection of built-in data types and control structures.

The following three subsections discuss peakedness and stability across inputs and versions of the receiver class distributions derived from profiles of these benchmarks.

3.3.1 Peakedness

For each language, we computed call-site-specific (1-CCP) class distributions and merged together call sites that exhibited the same degree of polymorphism (i.e., those that had the same number of distinct receiver classes at run-time). The results are shown in Figure 11. The height of a bar in each histogram indicates the dynamic percentage of sends (virtual function calls) in the benchmarks with that degree of polymorphism. Each bar of the histogram, reporting execution frequencies of sends with degree *N* polymorphism, is vertically divided into *N* parts showing the relative frequency from the most common to the least common receiver. Thus, the bottom shaded portions of the bars report the frequency with which messages were sent to the most common receiver class at that call site. The sends with polymorphism greater than 20 are collected together into the 20+ bin. These graphs show the aggregate results for all of the benchmarks in a particular language with each benchmark program weighted equally. (The same graphs drawn for each program in isolation have similar shapes.)

In both languages, the most common receiver class at a call site receives the majority of the messages sent at that call site; 71% of the C++ messages and 72% of the Cecil messages were sent to the most common receiver class. This indicates that predicting uniformly for the most likely receiver class at a given call site would lead to a success rate over 70%.

In C++, 36% of the dynamic dispatches occurred at call sites which only had a single receiver class, and 50% of the Cecil messages were sent at call sites with a single receiver class. Calder and Grunwald also studied the characteristics of class distributions of C++ programs and found that 91% of the messages were sent to the most common receiver class and that 66% of the call sites only sent to a single receiver class [Calder & Grunwald 94]. Our C++ programs are larger, and judging from the greater degree of polymorphism, programmed in a more object-oriented style. However, even in our programs there are still a large percentage of virtual function call sites with only a single receiver class.

3.3.2 Stability Across Inputs

To assess the cross-input stability of class distributions, we compared the profiles of each of our C++ and Cecil programs run on two different input sets. We attempted to make the inputs as different as possible to present a worst case scenario for the cross-input stability of receiver class distributions. We first present the results of our two interactive C++ programs, `doc` and `idraw`, which are the least stable of all the measured applications. We only present 0-CCP and 1-CCP stability for these programs, since our C++ profiling technology does not yet support gathering n-CCP distributions. We then present the stability results for the two Cecil applications, the Vortex compiler and a MIPS global instruction scheduler. These two programs, and the remainder of the C++ programs, are batch-oriented applications and have similar cross-input stability characteristics.

We use several metrics to evaluate the stability of receiver class distributions. One metric is the L_2 difference* between two normalized distributions, which is a very good indicator of high or low stability, but is not a particularly accurate metric for assessing two distributions that are somewhat similar. One advantage of this metric is that it allows an abstract comparison of two class distributions that is independent of any application of the distributions. We also use two additional metrics that are more directly related to our intended application of the class distributions. The FirstSame metric classifies two distributions as the same as long as their most common receiver classes are the same. Since the most common receiver class is usually the most important for class prediction, we expect that the FirstSame metric is a fairly realistic measure of stability for the purpose of guiding receiver class prediction. We also use a final, very stringent metric, OrderSame, which classifies two distributions as the same only if they are comprised of the same classes in the same frequency order.

Figure 12 presents cross-input stability results for the average of the two interactive C++ applications, using the abstract L_2 norm comparator. Figure 13 presents corresponding results for the average of the two batch Cecil applications (the batch C++ programs show similar characteristics to the batch Cecil programs). The receiver class profiles of the interactive C++ programs are fairly stable across inputs, and the receiver class profiles of the Cecil applications and of the batch-oriented C++ applications are even more stable.

Table 2 presents the results of applying the FirstSame and OrderSame metrics to measure cross-input stability. The numbers represent the dynamic percentage of class distributions considered the same by the metric in question.

Table 2: Cross-Input Stability Summary
% of distributions considered equivalent

	C++	Cecil
0-CCP, FirstSame	99%	99.9%
1-CCP, FirstSame	79%	99.9%
n-CCP, FirstSame	n/a	99.9%
0-CCP, OrderSame	28%	84%
1-CCP, OrderSame	45%	87%
n-CCP, OrderSame	n/a	88%

* If distributions with n receiver classes are considered as points in an n-dimensional space, the L_2 norm is the Euclidean distance between two points.

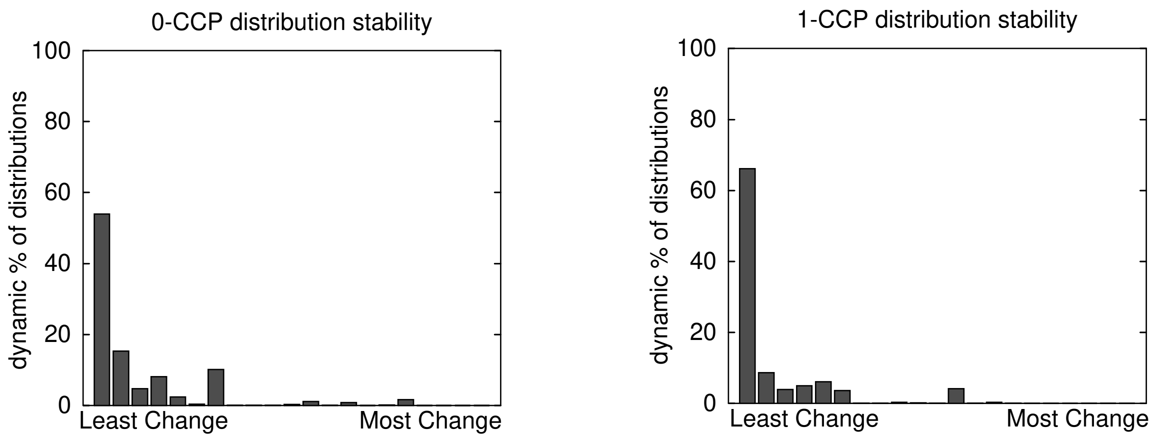


Figure 12: Interactive C++ Stability Across Inputs

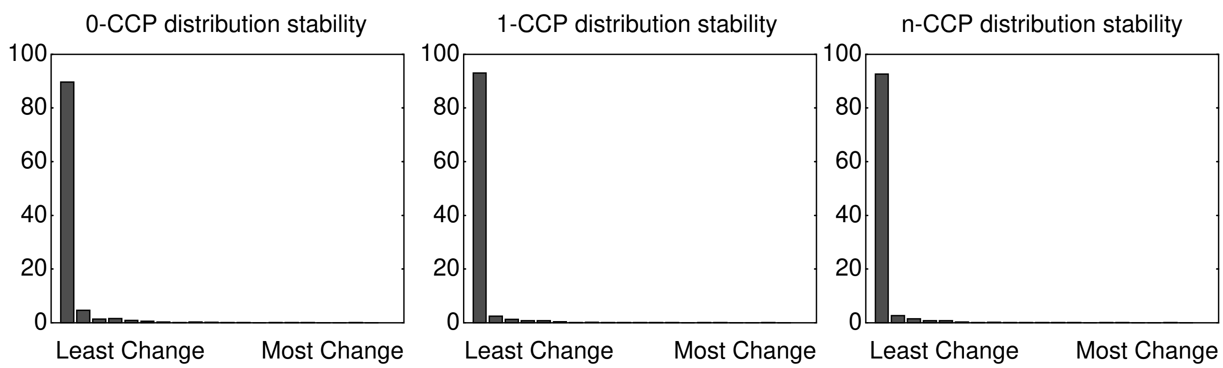


Figure 13: Cecil Stability Across Inputs

According to the FirstSame metric, class distributions were extremely stable across inputs. Even by the stringent OrderSame metric the Cecil distributions were very stable and the C++ distributions were somewhat stable. Since we believe that the FirstSame metric closely models how profile-guided class prediction uses receiver class distributions, this data indicates that distributions have enough cross-input stability to support this optimization.

3.3.3 Stability Across Program Versions

We would like class distributions to be stable across versions of a program undergoing rapid development. If this stability holds, then a profile from an older version of the program can be reused over many future versions without requiring reprofiling after each programming change. To determine the degree of cross-version stability exhibited by class distributions, we used RCS logs [Tichy 85] to recreate 12 snapshots in the development of the Vortex compiler (written in Cecil). The measured period began approximately one month after the compiler reached a basic, stable level of functionality (the compiler could optimize itself) with snapshots being taken twice a month for a period of six months. During this time, the application almost doubled in size, growing from 22,000 to 38,000 lines as additional optimization passes and other functionality were added. Support for selective recompilation was added, which had implications for many pieces of the compiler. Also during this period, many existing pieces of the compiler were substantially modified. The class hierarchies defining its internal representation were completely redesigned and a new

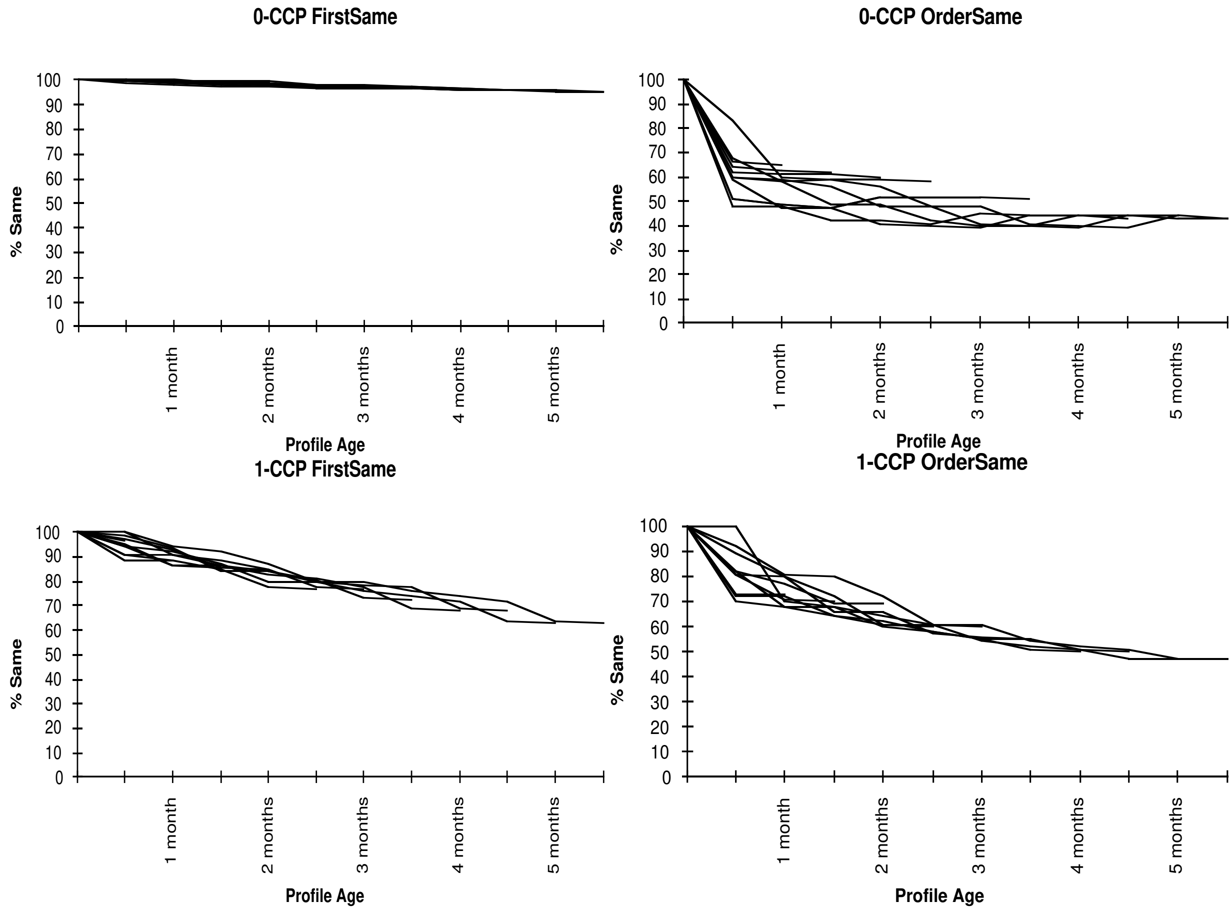


Figure 14: Cecil Stability Across Versions

iterative data flow analysis engine was added; many key data structures such as the method table and the compile-time method lookup cache were replaced with more efficient implementations. There were also pieces of the compiler that remained largely unchanged during this period; for example, only slight changes were made to the scanner and parser.

The results of our cross-version stability study are presented in Figure 14. Only 0-CCP and 1-CCP distributions are reported, since older versions of the system did not support collecting n-CCP distributions. As in the previous section, all data points represent dynamic frequencies. We compared each profile to every later profile, and plotted lines showing how each profile degraded in predictive quality over time. For example, the data points whose *x* coordinates are "1 month" represent the comparison of two profiles which were taken one month apart. There are 12 lines in each graph each one representing a single profile compared to all later profiles. According to the very stringent OrderSame metric, the distributions were somewhat unstable, since a significant percentage of the distributions changed in only a two-week period. However, according to the more realistic FirstSame metric, class distributions were quite stable. Applying this metric, fewer than 5% of the 0-CCP distributions changed over the entire six-month period, and it took around two months before more than 10% of the 1-CCP distributions changed. This suggests that old profiles are still quite useful for guiding receiver class prediction for future versions of a program. Section 5 reports empirical results confirming this suggestion.

Future work includes gathering similar revision histories for C++ applications to confirm the Cecil results.

3.4 Gathering Profile Data

Profiling an application to get receiver class distributions 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. The expense of profiling and the ease with which different granularities of profile data can be gathered depends in large part on the run-time system's message dispatching mechanism. Some systems, including Vortex and the implementations of Self, use *polymorphic inline caches* (PICs) [Hölzle et al. 91], which are call-site-specific association lists mapping individual receiver classes to target methods; for dispatching speed, the association list is represented as executable code. 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. Each PIC is annotated with as much inlined call chain context as is available.

Other systems, such as C++, rely on dispatch tables to direct messages. To gather profiles of C++ programs, we have been using the stop-gap measure of recognizing the assembly code generated by our C++ compiler for a virtual function call, and inserting instrumenting code that records call-site-specific distribution information (not call chain information), at relatively high run-time cost. We are switching to a Vortex-based C++ implementation that uses PICs to gather statistics about C++ virtual function calls when gathering profile data.

3.4.1 Profiling Optimized Code

To make profiling less intrusive, we wish to be able to profile optimized programs. In the presence of optimization, many dynamic dispatches that were present in the unoptimized program are no longer be performed, either because static analysis was able to determine the message target uniquely, or because receiver class prediction enabled the dynamic send to be avoided due to an earlier run-time class test. If an optimized program were profiled naively, the resulting distributions would be quite misleading for use in further optimization, since they would include none of the data for statically-bound calls or the commonly-occurring receiver classes.

We use two techniques to support accurate and efficient profiling of optimized programs. First, we do not instrument sends that were statically-bound solely due to static analysis. We reason that such sends will likely be able to be optimized similarly without resorting to profile guidance, and so do not need to have data recorded for them in the profile database. Consequently, we avoid introducing the significant cost of instrumenting such sends. Second, for sends optimized through class prediction, we instrument the successful branches as well as the dynamic send, combining their results into a single distribution representing the original unoptimized send. This preserves the high-frequency classes in the distribution accurately, for example allowing the compiler to detect when a common case becomes less common. The run-time overhead of this profiling for our fully-optimized Cecil benchmarks is in the range of 15% to 50%.

Much of this profiling overhead is due to very fine-grained profiling of certain commonly-occurring messages. For example, in Cecil (and Self and Smalltalk), `if` is implemented as a message to either `true` or `false`. This message is usually optimized through receiver class prediction, with tests for `true` and `false` being inserted before the `if` message to catch the common cases. (As class hierarchy analysis determines that there are no other possible implementations of `if`, the "otherwise" case is replaced by a call to the run-time system's message-not-understood error handler.) When profiling such a program, every `if` message will be instrumented to determine how many times `true` and `false` occurred, providing a kind of branch prediction information. However, counting the number of successful `true` and `false` class tests accounts for most of the run-time overhead of profiling in our system; languages with more traditional control and data models which do not monitor branch outcomes may observe a far lower cost for gathering receiver class distributions in optimized programs.

3.4.2 Iterated Optimization and Profiling

If a program has been optimized with receiver class prediction, profiling the optimized program can lead to better profile information than the profile of the unoptimized program. Some call sites will have been optimized with receiver class prediction, leading to additional inlining, which produces new call sites with longer inlined call chains. The profile of the optimized program may have multiple separate receiver class distributions for a call site in a method that has been inlined in multiple places, while the profile of the original unoptimized program has only one blended distribution. This process can be repeated again, profiling the second program optimized with the profile of the first optimized program, and so on, until no more improvements occur in the profile (i.e., when no more inlining is performed). Section 5 reports on the impact this successive reprofiling has on run-time performance.

4 Method Specialization

An important strategy in object-oriented design for increasing the reusability and malleability of code is factoring similar code out of abstract data type implementations and into shared superclasses. A single piece of source code then applies uniformly across a family of subclasses. To invoke subclass-specific behavior, the shared code includes dynamically-dispatched sends to the receiver formal parameter (`self`). Unfortunately, factoring can hurt run-time performance, by introducing extra procedure boundaries and more importantly by introducing new dynamically-dispatched sends with which the factored code regains access to specialized behavior.

The compiler can automatically undo the effects of factoring, without sacrificing its programmer benefits, by transparently producing specialized versions of shared code for inheriting subclasses. Several variations on method specialization have been developed, ranging from simple strategies like Self's customization to more sophisticated strategies like Vortex's profile-guided selective method specialization. The rest of this section discusses these techniques. Splitting, described in section 3.2.2, can be viewed as a kind of intraprocedural specialization.

4.1 Customization

Customization is a simple specialization scheme used in the implementations of several object-oriented languages, including Self [Chambers & Ungar 89, Hölzle & Ungar 94a], 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 `self`. Because sends to `self` tend to be fairly common in object-oriented programs, customization is effective at increasing execution performance: Self code runs 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 speedup, and argued for the inclusion of customization in C++ implementations [Lea 90].

Unfortunately, customization suffers from the twin problems of *overspecialization* and *underspecialization*, because specialization is done without considering its costs and benefits on a case-by-case basis:

- Overspecialization occurs whenever multiple specialized versions of a method are identical or nearly so, and could be coalesced into a single shared compiled method 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 an explosion in compiled code. 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 not likely to be practical.
- Underspecialization can occur with simple customization because methods are never specialized on arguments other than the receiver. In some cases, considerable benefits can arise from specializing a method for particular argument classes. On the other hand, indiscriminate specialization over multiple arguments would lead to a combinatorial explosion, explaining why previous systems customized only on the receiver argument.

In systems employing dynamic compilation, such as Self, 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.

4.2 Profile-Guided Selective Method Specialization

The Vortex compiler includes an alternative method specialization algorithm that exploits both class hierarchy information and dynamic profile information to reduce the problems of overspecialization and underspecialization. Rather than specializing exhaustively, Vortex is guided by dynamic profile data to selectively specialize only heavily-used methods for their most beneficial argument classes; the original general-purpose versions of methods is retained to handle the remaining cases. To avoid producing multiple similar specializations and to reuse existing specializations for the widest range of argument classes, Vortex uses class hierarchy information to identify the set of argument class combinations that can safely share a particular specialized method.

Our algorithm is driven by a weighted call graph derived from profile data. Each node represents a method in the profiled program. A given node has a set of call sites, and each call site has a set of outgoing call arcs pointing to the node representing the invoked method (multiple outgoing arcs for a single call site are possible for dynamically-dispatched call sites that invoke more than one callee at run-time). The weight of each call arc is given by the number of times it was traversed in the profiled execution.

The goal of the algorithm is to eliminate heavily-executed (high-weight) dynamically-dispatched call arcs by specializing the calling 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 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]). Our algorithm thus focuses on pass-through dynamically-dispatched call sites, which we term *specializable call sites*, since these are the sites that can most easily be determined to benefit from specialization. (More sophisticated jump functions are possible, although it quickly becomes difficult to accurately estimate the costs and benefits of specialization as inlining and other downstream effects are taken into account. Perhaps the inlining database described in section 2.1.3 would help.)

In summary, our algorithm visits each method in the call graph. The algorithm searches for high-weight, dynamically-dispatched, pass-through call arcs from the method, i.e., those call arcs that are both possible and profitable to optimize through specialization of the sending method. Using the class hierarchy of the program being compiled, 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. We represent a particular specialization of a method with *n* arguments as an *n*-tuple of sets of classes. When the algorithm completes, each method is associated with a set of *n*-tuples of class sets representing the general-purpose version plus some number of specializations identified as profitable by the algorithm. Once this first identification pass completes, a second pass produces the specialized compiled methods themselves. Regular static class analysis will automatically perform the desired optimizations if the formal parameters of the specialized version are annotated with the class sets given in the specialization *n*-tuple.

We will illustrate our algorithm with the class hierarchy and method definitions shown in Figure 15 and the profile-derived weighted call graph shown in Figure 16. The class hierarchy consists of nine 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`. (We will give method definitions with all formals, including `self`, explicit.) The shaded regions show the "equivalence classes" that all invoke the same implementation of `m` and `m2` (note that the illustrations in Figure 15 are two views of the same underlying inheritance hierarchy).

A more detailed pseudocode version of our algorithm is shown in Figure 17. Set operations on tuples are defined to operate pairwise on the tuple elements. The profile-derived weighted call graph is accessed through four helper functions over call arcs: 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 example, for the arc labelled α in Figure 16, *Caller*(α) is `A::m4`, *Callee*(α) is `B::m2`, *CallSite*(α) is the send `arg2.`***`m2`***`()` within `A::m4`, and *Weight*(α) is 550. The function *PassThroughArgs* gives the relationship between formal parameters of the sending method and actual arguments of a call site, expressed as a set of bindings of formal parameter index to actual parameter index; a pass-through arc must have a non-empty set of bindings. For instance, *PassThroughArgs*⟦`f2.foo(x,f1)` *in* `m(f1,f2)`⟧ = {<1→3>,<2→1>} (the index of the receiver is 1). The algorithm accesses information about the class hierarchy through the *ApplicableClasses* function. *ApplicableClasses*⟦*meth*⟧ returns the *n*-tuple of the set of classes for each formal argument for which the method *meth* could be invoked (excluding classes that bind to overriding methods); these are the "applies-to" tuples constructed for compile-time method lookup as described in section 2.2.1. The shaded regions in

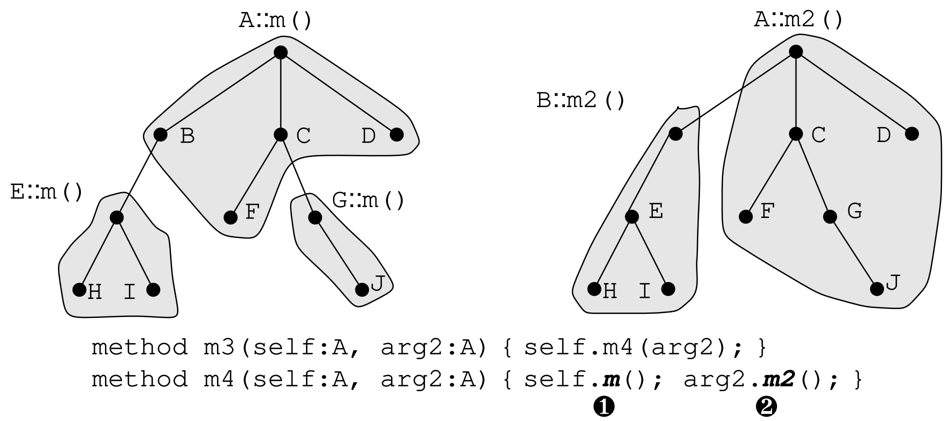


Figure 15: Example class hierarchy

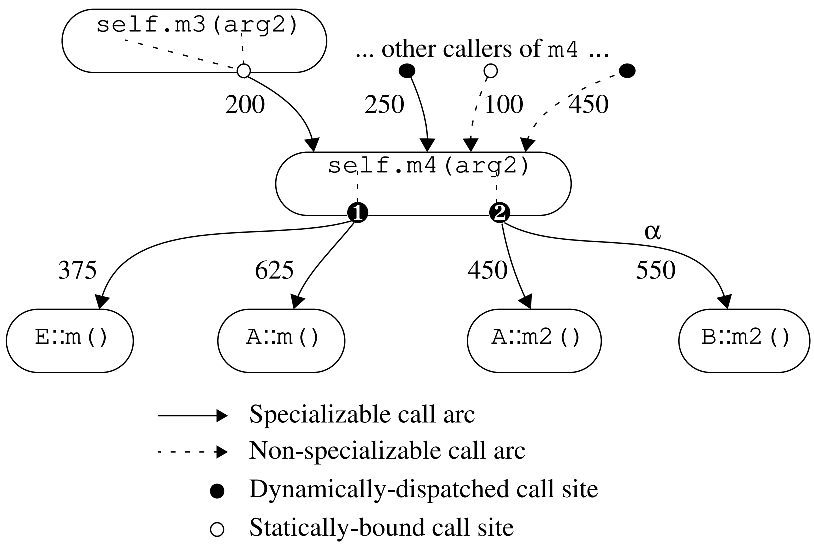


Figure 16: Example weighted call graph

Figure 15 visually identify the *ApplicableClasses* for each of the `m` and `m2` methods. For example, *ApplicableClasses*⟦`method E::m()`⟧ = <{E,H,I}>.

The remainder of this section examines several key aspects of our selective specialization algorithm in more detail:

- 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 an arc important to specialize? The algorithm currently uses a very simple heuristic, and Section 4.2.4 discusses the tradeoffs involved.

Helper functions:

Caller(a), *Callee*(a), *CallSite*(a), and *Weight*(a) give
caller, callee, call site, and execution count for arc a.

PassThroughArgs⟦`msg(arg`$_1$`, arg`$_2$`, ..., arg`$_n$`)` *in* `m(f`$_1$`, f`$_2$`, ..., f`$_m$`)`⟧=
$\{<fpos \rightarrow apos> \mid \mathtt{f}_{fpos} = \mathtt{arg}_{apos}\}$

ApplicableClasses⟦`method m(f`$_1$`, f`$_2$`, ..., f`$_n$`)`⟧ =
n-tuple of sets of classes for `f`$_1$`, f`$_2$`, ..., f`$_n$ for which `m` might be invoked

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

specializeProgram() =
  foreach method *meth* **do**
    $Specializations_{meth}$:= *ApplicableClasses*⟦*meth*⟧;
  foreach method *meth* **do**
    specializeMethod(*meth*);

specializeMethod(*meth*) =
  foreach *arc* s.t. *Caller*(*arc*) = *meth* **and** *isSpecializableArc*(*arc*) **do**
    if *Weight*(*arc*) > *SpecializationThreshold* **then**
      addSpecialization(*meth*, *neededInfoForArc*(*arc*));

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

neededInfoForArc(*arc*) **returns** Tuple[Set[Class]] =
  return *neededInfoForArc*(*arc*, *ApplicableClasses*⟦*Callee*(*arc*)⟧);

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

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

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

Figure 17: Selective Specialization Algorithm

- At run-time, how is the right specialized version chosen? This is discussed in Section 4.2.5.
- What is the interaction between the specialization algorithm and first-class nested functions? Section 4.2.6 considers this question.

This section concludes with a discussion of how specialization can be adapted to work in a system based on dynamic compilation.

4.2.1 Computing Specializations for Call Sites

The algorithm visits each high-weight pass-through arc leaving a method. 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 *ApplicableClasses* 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 *ApplicableClasses* is returned unchanged. As an example, consider arc α from the call graph in Figure 16. For this arc, the caller's *ApplicableClasses* is <{A,B,...,J},{A,B,...,J}>, the callee's *ApplicableClasses* 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` ($Specializations_{\mathtt{m4}}$).

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 "covers all the bases," producing method specializations for all plausible combinations of arc specializations. This 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 $<A_1,...,A_n>$ and $<A_1 \cap B_1,...,A_n \cap B_n>$, adding a new arc specialization tuple $<C_1,...,C_n>$ leads to four method specialization tuples for the method: $<A_1,...,A_n>$, $<A_1 \cap B_1,...,A_n \cap B_n>$, $<A_1 \cap C_1,...,A_n \cap C_n>$, and $<A_1 \cap B_1 \cap C_1,...,A_n \cap B_n \cap C_n>$ (assuming none of these intersections are empty: tuples containing empty class sets are dropped). For the example in Figures 15 and 16, nine versions of `m4` would be produced, including the original unspecialized version, assuming that all four outgoing call arcs were above threshold.

Although 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, we have not observed this behavior in practice. For the benchmarks described in Section 5, 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 have a small number of formal arguments that are highly polymorphic.

Were exponential blow-up 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. Of course, 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 the call site 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 a case, there are two choices: these statically-bound calls could be left unchanged, having them call the general-purpose version of the routine, or the statically-bound call could be replaced with a dynamically-bound call that selects 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 recursively 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 *specializedMethod* will attempt to optimize that arc. For example, when specializing the `m4` method for the tuple `<{A,B,C,D,F}, {A,C,D,F,G,J}>`, the arc from the `m3` method is identified as a target for cascaded specialization, but none of the other three callers are. 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. For this example, the computed class set tuple for `m3` (*callerSpec* in the algorithm) is `<{A,C,D,F}, {A,C,D,F}>`. 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 rising 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 to produce a specialized collection of mutually-recursive methods, other than the check to see whether the desired specialization has already been created. Recursive specialization stops when all callers are dynamically dispatched already or are not pass-through arcs or are of low weight. In effect, recursive specialization has the effect of hoisting dynamic dispatches out of high-weight parts of the call graph either to lower-weight places higher up in the call graph or to places where they can be combined with previously-existing dynamic dispatches.

4.2.4 Improved Cost-Benefit Analysis

The algorithm as presented currently uses a very simple heuristic for deciding what to specialize: if the weight of a specializable arc is larger than the *specializationThreshold* parameter, then the arc will be considered for specialization. In our implementation, the *specializationThreshold* is 1,000 invocations. There are several shortcomings of this simple approach. First, the code space increase incurred by specializing is not considered. A more intelligent heuristic could compute the set of specializations that would be necessary to statically bind a particular arc, and factor this information into the decision-making process. Second, it treats all dynamic dispatches as equally beneficial for specialization. Due to the indirect effects of optimizations such as inlining, the benefits of statically binding some message sends can be much higher than others. A more sophisticated heuristic could estimate the performance benefit of static binding, taking into account post-inlining optimizations. Third, the heuristic has no global view on the consumption of space during specialization. Alternatively, the algorithm could be provided with a fixed space budget, and could visit arcs in decreasing order of weight, specializing until the space budget was consumed. Nevertheless, as the results in section 5 show, our current simple heuristic incurs very low space cost for specialization. If anything, our current algorithm is not specializing aggressively enough.

4.2.5 Invoking the Appropriate Version of a 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) from the perspective of 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, including trees of single dispatch tables [Kiczales & Rodriguez 89], compressed multi-method dispatch tables [Chen et al. 94, Amiel et al. 94], and polymorphic inline caches extended to support multiple arguments [Hölzle et al. 91] as used by Vortex.

4.2.6 Specializing Nested Methods

In the presence of lexically-nested first-class functions, a message can be sent to a formal not of the outermost enclosing method but to a lexically-nested function (such as a closure that accepts arguments). Although our current implementation only considers the outermost method for specialization, in principle there is no reason that the algorithm could not produce multiple specialized versions of nested methods. At run-time, the representation of a closure could not contain a direct pointer to the target method's code, but instead would contain the address of a stub routine that performed the necessary dispatching for the specialized argument positions.

4.2.7 Applicability to a Dynamic Compilation Environment

Selective specialization is suitable for dynamic compilation environments such as Self as well as static compilation environments like Vortex. As described in section 3.2.1, the Self system initially compiles methods without optimization but installs counters to detect the heavily-used methods. When a counter exceeds a threshold, the system recompiles portions of the code using optimization to adapt the code to program hot spots. Our specialization algorithm could be used in such a system. The unoptimized code would keep track of the target methods for each call site and the counts (essentially arcs in the call graph), and the appropriate localized portion of the call graph could be constructed as necessary to make specialization decisions during the recompilation process. Section 5 provides results showing that, even for a system that compiles code lazily, our approach could lead to a significant reduction in code space requirements over a system that employs simple customization.

5 Performance Studies

Previous sections of this paper presented a number of analyses and transformations that have been developed to improve the performance of object-oriented applications. This section assesses their effectiveness by applying them to several applications written in Cecil, a dynamically-typed, purely object-oriented language [Chambers 93]. Section 5.1 describes the applications used in our performance studies and some details of our experimental set-up. Vortex transforms dynamically-dispatched message sends into statically-bound procedure calls through the use of five main techniques: intraprocedural class analysis, class hierarchy analysis, exhaustive class testing, profile-guided receiver class prediction, and selective specialization. Section 5.2 focuses on the impact of each of these techniques, both in isolation and in combination with the others. Finally, section 5.3 presents a series of supporting experiments that address related issues.

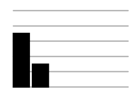
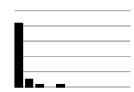
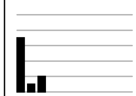
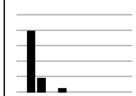
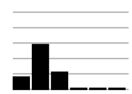
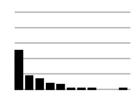
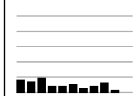
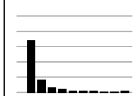
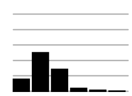
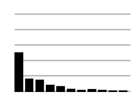
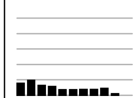
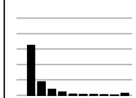
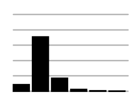
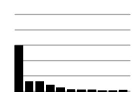
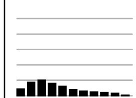
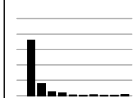
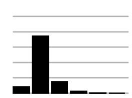
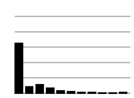
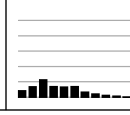
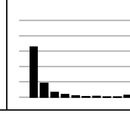
5.1 Methodology

To evaluate the performance of the techniques discussed in this paper, we have implemented them in the context of the Vortex compiler, and have selected five benchmark programs to compare the techniques, ranging in size from 600 to 75,000 lines. The benchmark programs are described in Table 3, along with some metrics about the use of inheritance and method overriding in the programs. These metrics are designed to give a feel for the programming style employed by these programs. In particular, note the deep and broad inheritance hierarchies and the large generic functions* for the larger programs.

The programs were compiled by Vortex into C code, which was then compiled using gcc-2.6.3 -O2 to produce an executable. To evaluate the techniques, we present data on both abstract program properties such as number of messages sent, as well as metrics specific to our particular implementation, such as execution time and compiled code space. All reported execution times are the median time from 11 runs of the application on an otherwise unloaded SPARCStation 20/61 with 128 MB of main memory running SunOS release 4.1.3_U1. With the exception of `richards` (which takes no input), different inputs were used to gather profile data and to measure application performance. Compiled code space numbers represent the size in bytes of a stripped program executable.

* All methods with the same name and number of arguments are considered to be part of a single generic function.

Table 3: Benchmark programs & their characteristics

Program	Description	Lines of Code	# of Classes	# of 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 generic functions consisting of *x* methods (x = 0..9,10+)
richards	Operating system simulation	644	30	96				
deltablue	Incremental constraint solver	2,831	81	584				
scheduler	MIPS global instruction scheduler	2,400 + 11,500 std. library	210	1587				
type-checker	Cecil typechecker	20,000 + 11,500 std. library	580	4447				
compiler	Vortex optimizing compiler	63,500 + 11,500 std. library	1201	9386				

5.2 Primary Results

In this section we examine the effectiveness of and the interactions between the five major techniques used by Vortex to eliminate message sends:

- intraprocedural class analysis, described in section 2.1.
- class hierarchy analysis, described in section 2.2.
- insertion of exhaustive tests, described in section 2.3.
- profile-guided receiver class prediction, described in section 3.2.
- selective specialization, described in section 4.2.

In Vortex, the implementations of class hierarchy analysis and profile-guided receiver class prediction are integrated with its implementation of intraprocedural class analysis. Therefore, we include intraprocedural class analysis in all the measured configurations.* Similarly, since selective specialization utilizes class hierarchy information, we do not consider selective specialization without also including class hierarchy analysis. Thus, the applications were compiled by Vortex using the following compiler configurations:

- **unopt** - No high-level optimizations for message sends.
- **intra** (or **i**)- Intraprocedural class analysis, automatic inlining, hard-wired class prediction for primitive message names such as + and `if`, extended splitting (Section 3.2.2), partial dead code elimination for

* Neither of these two techniques are inherently dependent on intraprocedural class analysis; the Self-93 system [Hölzle & Ungar 94a] incorporated profile guided receiver class prediction without intraprocedural class analysis, and in statically-typed object-oriented languages the static type of a message's receiver could be used to enable class hierarchy analysis without additional analysis.

closures (Section 2.1.4), value-based analysis (Section 2.4.1), elimination of dead and redundant load/stores (Section 2.4.2), and several standard iterative dataflow optimizations such as common subexpression elimination and dead assignment elimination.

- **i+CHA** - intra augmented with class hierarchy analysis (Section 2.2).
- **i+CHA+exh** - i+CHA augmented with exhaustive class testing for call sites that have a small but non-singular number of candidate methods (Section 2.3).
- **intra+CHA+exh+spec** - i+CHA+exh augmented with selective specialization (Section 4.2).

We consider each of these configurations both with and without profile-guided class prediction (Section 3). Table 4 contains data on the execution speed, the dynamic numbers of dynamically-dispatched message sends and class tests (comprising both singleton class and cone tests) executed, and compiled code space, for each of the optimization configurations for each of the benchmark programs. In addition to the raw data, this table also contains values normalized to the intra without profile data configuration of each program. Many of the tables in this section include normalized values, which are indicated by enclosing them in parentheses.

Table 4: Impact of message send optimizations

Program		Configuration	Execution time (secs)	Message sends executed	Class tests executed	Code space (bytes)
Richards	No profile data	unopt	13.76 (6.78)	68,781,098 (2.02)	570,705 (0.05)	1,078,584 (1.08)
		intra	2.03 (1.00)	34,089,777 (1.00)	12,046,388 (1.00)	999,072 (1.00)
		i+CHA	0.61 (0.30)	8,708,179 (0.26)	11,944,000 (0.99)	943,992 (0.94)
		i+CHA+exh	0.48 (0.24)	1,294,143 (0.04)	18,408,802 (1.53)	973,744 (0.97)
		i+CHA+exh+spec	0.45 (0.22)	1,345,337 (0.04)	16,782,463 (1.39)	982,056 (0.98)
	Profile data	intra	0.43 (0.21)	5,313,897 (0.16)	23,623,702 (1.96)	1,024,504 (1.03)
		i+CHA	0.33 (0.16)	3,404,726 (0.10)	16,105,103 (1.34)	952,872 (0.95)
		i+CHA+exh	0.35 (0.17)	723,861 (0.02)	18,306,423 (1.52)	981,888 (0.98)
		i+CHA+exh+spec	0.32 (0.16)	775,055 (0.02)	16,680,084 (1.38)	990,200 (0.99)
Deltablue	No profile data	unopt	53.13 (3.23)	329,334,346 (1.65)	13,126,308 (0.20)	1,097,312 (1.08)
		intra	16.46 (1.00)	199,620,412 (1.00)	64,031,797 (1.00)	1,017,888 (1.00)
		i+CHA	11.21 (0.68)	91,879,532 (0.46)	57,131,359 (0.89)	963,688 (0.95)
		i+CHA+exh	8.69 (0.53)	49,182,401 (0.25)	79,364,349 (1.24)	1,009,408 (0.99)
		i+CHA+exh+spec	6.89 (0.42)	38,433,971 (0.19)	67,844,863 (1.06)	1,009,568 (0.99)
	Profile data	intra	1.52 (0.09)	5,405,005 (0.03)	81,396,112 (1.27)	1,099,128 (1.08)
		i+CHA	1.38 (0.08)	2,432,206 (0.01)	72,754,305 (1.14)	1,017,408 (1.00)
		i+CHA+exh	1.58 (0.10)	931,811 (0.00)	82,783,339 (1.29)	1,053,288 (1.03)
		i+CHA+exh+spec	1.56 (0.09)	931,811 (0.00)	82,783,339 (1.29)	1,053,400 (1.03)

Table 4: Impact of message send optimizations

Program		Configuration	Execution time (secs)	Message sends executed	Class tests executed	Code space (bytes)
Scheduler	No profile data	unopt	21.11 (2.18)	9,925,584 (1.75)	412,079 (0.18)	2,144,128 (1.11)
		intra	9.69 (1.00)	5,662,957 (1.00)	2,306,070 (1.00)	1,936,032 (1.00)
		i+CHA	6.58 (0.68)	2,577,346 (0.46)	2,031,323 (0.88)	1,826,712 (0.94)
		i+CHA+exh	5.32 (0.55)	1,599,429 (0.28)	2,506,140 (1.09)	1,956,088 (1.01)
		i+CHA+exh+spec	4.98 (0.51)	1,401,864 (0.25)	2,271,423 (0.98)	2,029,040 (1.05)
	Profile data	intra	3.11 (0.32)	466,123 (0.08)	2,296,420 (1.00)	2,047,688 (1.06)
		i+CHA	2.61 (0.27)	398,716 (0.07)	2,008,608 (0.87)	1,921,064 (0.99)
		i+CHA+exh	2.74 (0.28)	377,142 (0.07)	2,215,137 (0.96)	2,008,728 (1.04)
		i+CHA+exh+spec	2.76 (0.28)	361,968 (0.06)	2,152,962 (0.93)	2,081,200 (1.07)
Typechecker	No profile data	unopt	333.82 (3.23)	117,898,851 (1.89)	1,576,558 (0.09)	5,176,632 (1.15)
		intra	103.44 (1.00)	62,357,301 (1.00)	18,238,949 (1.00)	4,519,096 (1.00)
		i+CHA	64.79 (0.63)	22,382,240 (0.36)	15,768,735 (0.86)	4,159,944 (0.92)
		i+CHA+exh	50.55 (0.49)	13,712,846 (0.22)	20,953,271 (1.15)	4,440,144 (0.98)
		i+CHA+exh+spec	46.49 (0.45)	11,031,174 (0.18)	22,698,360 (1.24)	4,594,656 (1.02)
	Profile data	intra	33.84 (0.33)	6,961,025 (0.11)	28,736,043 (1.58)	5,090,344 (1.13)
		i+CHA	30.08 (0.29)	6,409,824 (0.10)	20,360,226 (1.12)	4,578,232 (1.01)
		i+CHA+exh	29.78 (0.29)	5,605,325 (0.09)	21,631,091 (1.19)	4,735,416 (1.05)
		i+CHA+exh+spec	29.87 (0.29)	5,521,203 (0.09)	21,429,207 (1.17)	4,879,536 (1.08)
Compiler	No profile data	unopt	3,617.00 (2.41)	1,097,783,920 (1.78)	32,287,226 (0.20)	11,202,072 (1.15)
		intra	1,500.00 (1.00)	617,004,841 (1.00)	158,966,133 (1.00)	9,754,032 (1.00)
		i+CHA	903.00 (0.60)	246,662,311 (0.40)	132,188,864 (0.83)	9,825,936 (1.01)
		i+CHA+exh	700.00 (0.47)	153,558,538 (0.25)	198,647,910 (1.25)	9,985,384 (1.02)
		i+CHA+exh+spec	639.00 (0.43)	129,811,734 (0.21)	179,658,892 (1.13)	10,302,752 (1.06)
	Profile data	intra	615.00 (0.41)	124,908,520 (0.20)	273,418,374 (1.72)	10,708,512 (1.10)
		i+CHA	515.00 (0.34)	90,153,322 (0.15)	175,309,100 (1.10)	9,584,728 (0.98)
		i+CHA+exh	486.00 (0.32)	71,685,978 (0.12)	201,732,501 (1.27)	10,343,920 (1.06)
		i+CHA+exh+spec	506.00 (0.34)	71,587,088 (0.12)	196,963,036 (1.24)	10,596,760 (1.09)

Perhaps the first thing that one should notice about this data is that the impact of all the optimizations is exaggerated for the two small benchmarks, richards and deltablue. Since these small benchmarks tend to overstate the potential benefits of applying these techniques to larger, more realistic, programs, the discussion in the remainder of this section will focus on the three larger benchmark programs: scheduler, typechecker and compiler. The graphs in Figure 18 illustrate the average execution time speed-up and the average dynamic number of dispatches executed for these three larger programs, normalized to the intra configuration.

There are several major jumps in performance that are worth noting. First is the factor of 2.5 performance improvement (highlighted by the ① in the figure) and the corresponding 45% drop in the number of message

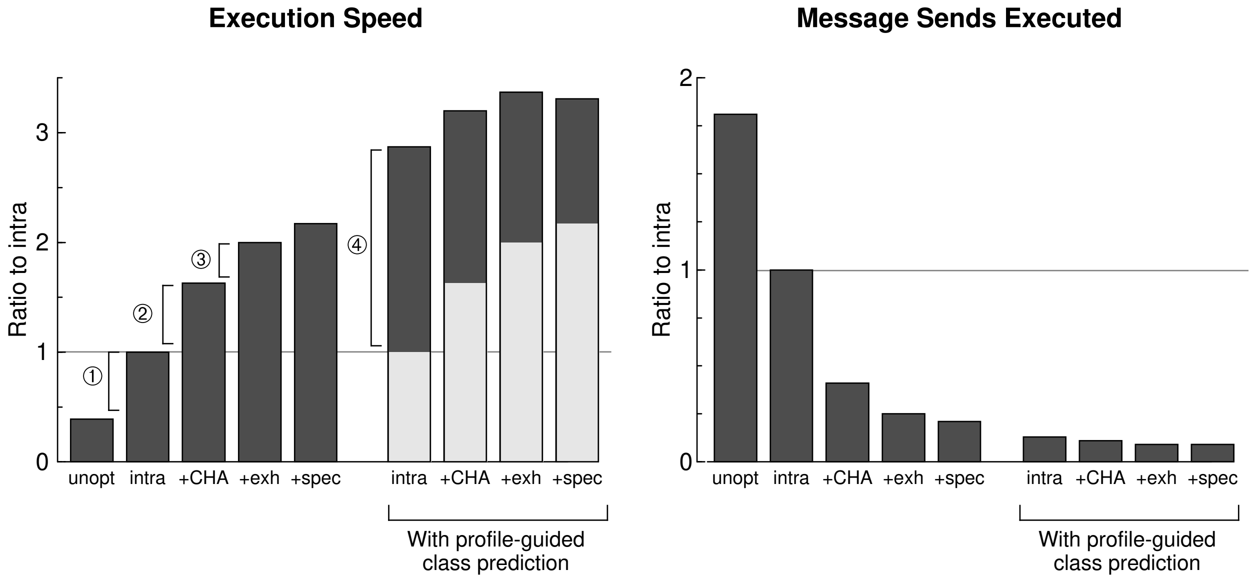


Figure 18: Effects of optimizations on three larger benchmarks

sends from unopt to intra. Most of this benefit is attributable to simple hardwired class prediction to eliminate the message passing overhead for common messages such as if and +. For purely object-oriented languages such as Cecil that use message sending to implement basic operations and control structures, an important initial step towards an efficient implementation is to optimize these messages. The second major performance improvement (②) comes from adding class hierarchy analysis on top of intraprocedural class analysis, giving about a 60% improvement in performance and a 60% reduction in the number of message sends over intra. The addition of exhaustive class testing further improves performance by 18% (③), and reduces the number of messages sent by nearly 40% again. Selective specialization, based on profile-derived execution frequencies, improves performance by about 8% over class hierarchy analysis and exhaustive class testing. The impact of selective specialization was larger in the past [Dean et al. 95], before we developed and implemented the exhaustive class testing algorithm. However, exhaustive class testing for small numbers of candidate methods and the selective specialization algorithm are both optimizing the same kinds of message sends, and so adding specialization on top of exhaustive class testing provides relatively little performance improvement.

The substantial impact of profile-guided class prediction is perhaps the most noticeable trend in the graph, where the execution speed improvement attributable to profile-guided class prediction is represented by the darker shaded portions of the bar (e.g. ④). When combined with a relatively weak static analysis like simple intraprocedural class analysis, profile-guided class prediction provides a factor of 2.8 performance improvement, along with an 87% reduction in the number of message sends. As the compiler applies more powerful static analyses, the performance improvement due to profile-guided class prediction decreases. For example, for a system that applies class hierarchy analysis and exhaustive class testing, the performance improvement due to profile data is 68%. More powerful interprocedural class analysis techniques might further narrow the performance gap between a purely static and a static+profile system [Grove 95, Agesen & Hölzle 96]. Although profile-guided class prediction has the largest individual impact of any of these techniques, the 18% performance improvement of i+CHA+exh+profile over i+profile, indicates that the static guarantees provided by static analyses give performance benefits over the “likely but not guaranteed” information provided by profile-guided class prediction. Thus, there is a synergy between static techniques like class hierarchy analysis and exhaustive class testing and dynamic techniques like profile-guided class prediction, indicating that to achieve the highest performance, compilers should apply both kinds of information.

Considering the large amount of inlining that is being performed as a result of these optimizations, the impact on code space is surprisingly small. In part this is because message sends in our implementation require a significant amount of code so that the inlined code often ends up being smaller than the code for an equivalent message send. For all but one of our benchmarks the unopt version had the largest executable of any of the configurations. Profile-guided class prediction tended to increase the code space requirements by about 10% over an equivalent system without profile data. The compile time impact of these optimizations is more significant in our system. Moving from unopt to intra increases compile time by about 40%. Adding any or all of the class hierarchy analysis, exhaustive class testing, or profile-guided class prediction optimizations further increases compile time by about 70%. Most of the additional compile time comes as a result of performing additional inlining (and subsequent optimizations of the inlined code), not in performing the analyses themselves. These compilation time results should not be scrutinized too closely, since our compiler is a research infrastructure designed primarily for ease of extension and experimentation, and not for short compile times.

5.3 Supporting Experiments

The previous section presented an overall assessment of the primary optimization techniques utilized by Vortex to reduce message passing overhead. This section presents a series of additional experiments that focus on a number of interesting secondary questions. Subsequent subsections address the following issues:

- The benchmark applications are all Cecil programs. The overall efficiency of the language implementation may have a substantial impact on the magnitude of the expected runtime benefits of the studied optimizations. How does the absolute performance of Cecil compare to other languages such as C++, Self, and Smalltalk?
- Section 2.1.4 described how Vortex uses partial dead code elimination to delay closures allocation operations until they are absolutely necessary, in the hope of removing them from the main control flow paths. How effective is this technique at reducing the dynamic number of closure allocations and how much impact does this have on application performance?
- Vortex incorporates several interrelated analyses that it uses to optimize instance variable accesses by detecting and eliminating redundant and dead loads and stores (section 2.4.2). How often are these optimizations applicable, and what impact do they have on application performance?
- Receiver class prediction can introduce partially redundant class tests when multiple messages are sent to the same receiver. Splitting, described in section 3.2.2, eliminates redundant class tests by duplicating portions of the control flow graph. What are the costs and benefits of this transformation?
- The effectiveness of profile-guided receiver class prediction is highly dependent on the predictive quality of the profile-derived class distributions that drive it. The stability and peakedness results from section 3.3 suggest that predictiveness is directly related to the degree of context associated with the class distribution. The profiles used in the experiments in section 5.2 were iterated nCCP profiles, and thus had a large degree of associated context. To what extent does lack of context degrade the effectiveness of profile-guided receiver class prediction?

- In order for off-line profile-guided receiver class prediction to be practical in a program development environment, we must be able to avoid re-profiling the application after each program change. Ideally, a single profile could be utilized across a large number of programing changes. How much does the predictiveness of the profile-derived class distributions degrade as the application evolves?
- Section 5.2 reported on the performance impact of selective method specialization. How does the performance of selective specialization compare with that of method customization (described in section 4.2)?

For all of these experiments, the base configuration is the i+CHA+exh+spec with profile-guided class prediction configuration (our highest optimizing configuration).

5.3.1 Absolute Cecil Performance

We first quantify the absolute performance of Vortex-compiled Cecil programs. We compare Vortex's implementation of Cecil to optimizing implementations of C++, a statically-typed, hybrid object-oriented language, and Smalltalk [Goldberg & Robson 83] and Self [Ungar & Smith 87], two dynamically-typed, purely object-oriented languages. The comparison between Cecil and C++ quantifies the performance gap introduced by Cecil's more expressive language features. The comparison to Self is interesting both because Self and Cecil have a number of advanced language features in common and because the Self 4.0 system is a carefully-engineered state-of-the-art implementation that outperforms many commercial Smalltalk systems [Hölzle & Ungar 94a].

- Cecil: Vortex in the base configuration. This is the compiler configuration we use in our day to day development of the Vortex compiler.
- Smalltalk: VisualWorks 2.0, a commercial Smalltalk implementation by ParcPlace Systems, Inc.
- Self: Self 4.0 performing type-feedback, customization, intraprocedural class analysis, and splitting [Hölzle & Ungar 94a]
- C++: GNU g++ 2.6.3 -O2 -finline-functions

Unfortunately, currently the only non-trivial programs written in all four languages are `deltablue` and `richards`. Although the execution times shown in Table 5 may be indicative of the relative performance of larger applications, no definitive conclusions can be made until several large applications are available in all four languages. Conclusions based solely on small benchmarks can be misleading, since relatively

Table 5: Cecil performance relative to other languages

Benchmark	Cecil	Smalltalk[a]	Self	C++
richards	0.32 sec	0.98 sec	0.23 sec	0.10 sec
deltablue[b]	1.56 sec	5.55 sec	2.62 sec	0.34 sec

a. Smalltalk times are elapsed (wall clock) time, all other times are CPU times. We observed a 10-15% difference between elapsed time and CPU time for the Cecil versions of these two programs.
b. chainTest benchmark with input parameters 50,500.

minor changes can have a large impact on observed execution times. For example, Cecil's inlining heuristics are different (and possibly slightly less aggressive) than those used by Self 4.0. Thus, at one of the call-sites inside the critical method of the richards benchmark Self chose to inline the invoked method and Cecil did not. By adding a single `inline` pragma to the Cecil version of the benchmark, thus forcing just one call-site to be inline expanded instead of being left as an out-of-line call, the Cecil execution time dropped to 0.28 seconds, cutting the performance gap between the Self and Cecil implementations of richards in half. However, this data does suggest that the four language implementations can be divided into three groups: C++, Cecil and Self, and Smalltalk. It appears that Cecil and Self have roughly equivalent performance, that Cecil and Self are roughly a factor 3 to 4 slower than C++, and that Smalltalk is about 3 to 4 times slower than Cecil and Self.

5.3.2 Partial Dead Code Elimination for Closures

Section 2.1.4 described how partial dead code elimination can be adapted to push infrequently-needed closure allocation operations off the main control flow path by delaying the allocation until it is actually needed. In aggressive optimizing implementations of languages such as Cecil and Self that utilize closures heavily, this optimization may be significant because many closures will be inline-expanded along a subset of the control flow paths through the method, thus obviating the need to ever create the closure object. We quantify the impact of this optimization on application performance by comparing the base configuration to one that does not perform this optimization. Table 6 shows the execution speeds and the dynamic number of closure objects created for each application with and without this optimization.

Table 6: Impact of partial dead code elimination for closures

Program	Base		Base without closure delaying	
	Execution time (secs)	Closures created	Execution time (secs)	Closures created
richards	0.32 (1.00)	66 (1.00)	1.11 (3.47)	4,031,555 (61,084.17)
deltablue	1.56 (1.00)	33,036 (1.00)	4.82 (3.09)	14,886,666 (450.62)
scheduler	2.76 (1.00)	173,758 (1.00)	3.73 (1.35)	560,207 (3.22)
typechecker	29.87 (1.00)	703,296 (1.00)	39.13 (1.31)	3,824,148 (5.44)
compiler	506.00 (1.00)	14,320,530 (1.00)	572.00 (1.13)	38,614,425 (2.70)

The results show that without this optimization, there is a a dramatic increase in the dynamic number of closures creations (a factor of several hundred or thousand for the two small benchmarks, and between a factor of 3 and 6 for the larger benchmarks), and that this increase has a substantial impact on program execution time (more than a factor of 3 slowdown for the two smaller benchmarks, and between 13% and 35% for the larger benchmarks). As in Section 5.2, the small benchmarks tend to exaggerate the impact of the optimization.

5.3.3 Eliminating Dead and Redundant Memory Operations

In section 2.4.2 we presented a series of simple optimizations for instance variable accesses that eliminate redundant or dead loads and stores of instance variables and can even completely eliminate object allocations, effectively by allocating the object's instance variables to local variables. To quantify the impact of these techniques, we compare the execution times, the dynamic number of instance-variable-related loads and stores, and the dynamic number of objects created in programs compiled in the base configuration and one that does not perform these optimizations. The results of this comparison are shown in Table 7

Table 7: Impact of dead and redundant memory operation elimination

Program	Base			Base without memory optimizations		
	Execution time (secs)	Objects created	Instance variable loads/ stores	Execution time (secs)	Objects created	Instance variable loads/ stores
richards	0.32 (1.00)	892 (1.00)	13,058,441 (1.00)	0.35 (1.09)	892 (1.00)	13,756,666 (1.05)
deltablue	1.56 (1.00)	914,652 (1.00)	49,115,347 (1.00)	1.55 (1.00)	914,652 (1.00)	53,924,443 (1.10)
scheduler	2.76 (1.00)	433,074 (1.00)	1,205,868 (1.00)	2.94 (1.07)	433,074 (1.00)	1,331,841 (1.10)
typechecker	29.87 (1.00)	1,976,969 (1.00)	18,112,976 (1.00)	30.48 (1.02)	1,976,969 (1.00)	19,515,496 (1.08)
compiler	506.00 (1.00)	26,736,310 (1.00)	160,309,842 (1.00)	505.00 (1.00)	26,747,232 (1.00)	179,244,852 (1.12)

On average, the optimizations are able to eliminate about 10% of instance variable operations as either dead or redundant. The elimination of these memory operations can, in principle, cause object creations to become unnecessary. However, in practice, this seems to never happen: object creations were only eliminated in the compiler benchmark, and less than 0.05% of its object creations were removed. The impact on execution time of the optimizations is less than 10% for all the benchmarks, and is negligible for the compiler and typechecker benchmarks. However, it is possible that these optimizations may have greater value in the future. The effectiveness of our analysis is hampered because it is only intraprocedural; any pieces of code that are not inlined in a particular context require extremely conservative assumptions. We suspect that a more powerful interprocedural analysis would be able to prove that more object creations and instance variable loads and stores are dead or redundant. Interprocedural escape analysis might also be able to prove that many object creations can outlive their enclosing context, which would permit the objects to be allocated on the stack rather than being heap-allocated [Hudak 86, Kranz et al. 86].

5.3.4 Splitting

Splitting, described in section 3.2.2, eliminates potentially redundant class tests by duplicating portions of the control flow graph. To assess the costs and benefits of this technique, we measured the execution time, dynamic number of class tests, and compiled code size of applications compiled in the base configuration and one that does not perform splitting.

As shown in Table 8, splitting has a substantial impact on the number of class tests that are executed. Without splitting, the programs execute 51-96% more class tests, and for the two smallest benchmarks, the performance impact of splitting is substantial, at 22% for richards and 19% for deltablue. The performance impact on the larger benchmarks was much smaller, at between 3-4%. Splitting had virtually no overall impact on code space, indicating that the increases caused by duplicating code to preserve information were roughly offset by the decreases in code space enabled by the elimination of class tests and branches.

5.3.5 Profile Data and Context

The effectiveness of profile-guided receiver class prediction is highly dependent on the predictiveness of the profile-derived class distributions that drive it. As discussed in section 3.1.1, increased context can increase the predictive ability of the profile data by removing imprecisions introduced by polymorphism. The amount of context available to the compiler can be increased by iterating the process of profiling and optimizing an application (see section 3.4.2). To quantify the importance of these effects, we compiled the applications with the following configurations:

Table 8: Impact of splitting

Program	Base			Base without splitting		
	Execution time (secs)	Class tests executed	Code space	Execution time (secs)	Class tests executed	Code space
richards	0.32 (1.00)	16,680,084 (1.00)	990,200 (1.00)	0.39 (1.22)	25,234,223 (1.51)	998,216 (1.01)
deltablue	1.56 (1.00)	82,783,339 (1.00)	1,053,400 (1.00)	1.86 (1.19)	129,095,582 (1.56)	1,060,648 (1.01)
scheduler	2.76 (1.00)	2,152,962 (1.00)	2,081,200 (1.00)	2.84 (1.03)	4,219,834 (1.96)	2,068,376 (0.99)
typechecker	29.87 (1.00)	21,429,207 (1.00)	4,879,536 (1.00)	31.16 (1.04)	41,439,823 (1.93)	4,896,904 (1.00)
compiler	506.00 (1.00)	196,963,036 (1.00)	10,596,760 (1.00)	522.00 (1.03)	367,553,408 (1.87)	10,605,472 (1.00)

- **static** - i+CHA+exh, as described in Section 5.2, which performs intraprocedural class analysis, class hierarchy analysis, and exhaustive class testing, forms the baseline for this set of experiments.
- **static+0CCP** - static with profile data restricted to 0-CCP (message summary) distributions
- **static+1CCP** - static with profile data restricted to 1-CCP (call-site) distributions
- **static+nCCP0** - static using all available call-chain context in profile data derived from the static version of the application.
- **static+nCCP1** - static using all available call-chain context in profile data derived from the static+nCCP0 version of the application.
- **static+nCCP2** - static using all available call-chain context in profile data derived from the static+nCCP1 version of the application.
- **static+nCCP3** - static using all available call-chain context in profile data derived from the static+nCCP2 version of the application.

Additional context has an impact only when a program exhibits different behavior in different contexts. The two smallest benchmarks, deltablue and richards, do not exhibit interesting behavior in this regard, and their results indicate that the degree of context present in profile data is unimportant. However, larger programs tend to exhibit interesting behavior depending on the amount of context present in the profile. Table 9 reports the execution speeds and dynamic number of class tests and dynamic dispatches for the three larger benchmarks for each of these configurations.

Table 9: Impact of additional context for profile data

Program	Configuration	Execution time (secs)	Class tests executed	Message sends executed
Scheduler	static	5.32 (1.00)	2,506,140 (1.00)	1,599,429 (1.00)
	static+0CCP	5.23 (0.98)	2,540,771 (1.01)	1,494,032 (0.93)
	static+1CCP	3.28 (0.62)	2,310,123 (0.92)	561,478 (0.35)
	static+nCCP0	2.73 (0.51)	2,222,223 (0.89)	393,136 (0.25)
	static+nCCP1	2.71 (0.51)	2,215,602 (0.88)	377,212 (0.24)
	static+nCCP2	2.81 (0.53)	2,215,137 (0.88)	377,142 (0.24)
	static+nCCP3	2.74 (0.52)	2,215,137 (0.88)	377,142 (0.24)

Table 9: Impact of additional context for profile data

Program	Configuration	Execution time (secs)	Class tests executed	Message sends executed
Typechecker	static	50.55 (1.00)	20,953,271 (1.00)	13,712,846 (1.00)
	static+0CCP	44.25 (0.88)	19,501,900 (0.93)	10,534,243 (0.77)
	static+1CCP	32.10 (0.64)	22,031,925 (1.05)	6,023,661 (0.44)
	static+nCCP0	31.13 (0.62)	21,676,996 (1.03)	5,949,521 (0.43)
	static+nCCP1	29.62 (0.59)	21,669,020 (1.03)	5,627,826 (0.41)
	static+nCCP2	29.94 (0.59)	21,635,705 (1.03)	5,611,544 (0.41)
	static+nCCP3	29.78 (0.59)	21,631,091 (1.03)	5,605,325 (0.41)
Compiler	static	700.00 (1.00)	198,647,910 (1.00)	153,558,538 (1.00)
	static+0CCP	671.00 (0.96)	237,453,342 (1.20)	137,338,299 (0.89)
	static+1CCP	497.00 (0.71)	210,370,092 (1.06)	76,258,480 (0.50)
	static+nCCP0	497.00 (0.71)	202,555,878 (1.02)	73,705,286 (0.48)
	static+nCCP1	507.00 (0.72)	201,970,643 (1.02)	72,368,420 (0.47)
	static+nCCP2	488.00 (0.70)	201,697,290 (1.02)	71,713,481 (0.47)
	static+nCCP3	486.00 (0.69)	201,732,501 (1.02)	71,685,978 (0.47)

The degree of context present in profile data does indeed have a significant effect on both the number of messages sent and on execution time. Message-level (0CCP) profile data performs much worse than does call-site specific (1CCP) profile data. Programs compiled with 1CCP information sent 43-62% fewer messages and ran 26-37% faster than they did when compiled with 0CCP information. Similarly, the additional context provided by the longer call chains in nCCP0 profile data sometimes makes a significant difference in both the number of messages sent and in bottom-line performance. For example, the scheduler benchmark executed 30% fewer message sends and ran 17% faster when compiled with nCCP0 profile information than when compiled with 1CCP profile data. The improvements due to additional call-chain context seem to be subject to the law of diminishing marginal returns. Iterating the compile-profile cycle once to produce nCCP1 profile data reduced the number of message sends by an additional 2-5%, but with little effect on execution time. Further iteration to produce longer call chain contexts reduced the number of messages sent only slightly and had virtually no effect on the execution time. Interestingly, the effects of iterating the compile-profile cycle to gather additional context information used to be more significant in our system, prior to the introduction of the exhaustive class testing algorithm [Grove et al. 95].

5.3.6 Old Profile Data

To measure the degree to which there was sufficient cross-version stability to allow the use of old profiles to optimize newer versions of the same program, we used the 1-CCP distributions from each of the profiled dates to optimize the compiler benchmark program sources from the most recent date (section 3.3.3 presents the experiments that generated these dated profiles). Figure 19 plots the execution speeds of the compiler program optimized with intraprocedural class analysis and class hierarchy analysis* and with profile data of varying ages (dated profiles) against that of a compiler program optimized with the same static optimizations but without profile data (i+CHA). Clearly, even using fairly inaccurate profile information is much better than using no profile data at all. For this application, it appears that profiling once

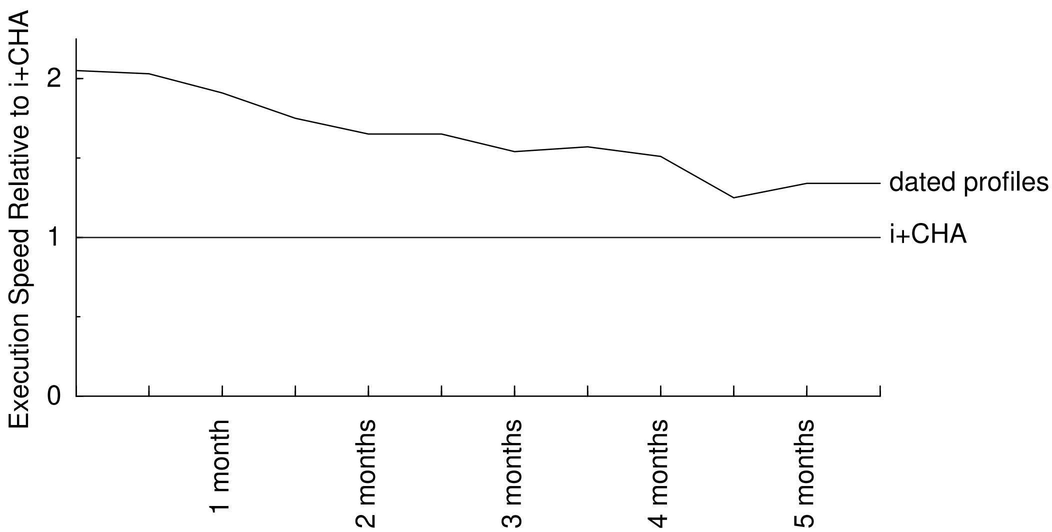


Figure 19: Effectiveness of Receiver Class Prediction Using Old Profiles

every 4 to 6 weeks would have been sufficient to maintain execution speeds that were within 90% of those enabled by completely up-to-date profiles.

5.3.7 Specialization vs. Customization

One way to obtain additional static information about the class of the arguments of a method is to compile multiple versions of the routine, each specialized to a different set of classes. The key issue of deciding what routines to specialize and for what argument classes impacts performance as well as code space and compile time. One approach, called customization [Chambers & Ungar 89], compiles a different specialized version of a routine for each class that inherits the routine, and is used in the compilers for Self [Chambers & Ungar 89, Hölzle & Ungar 94a], Sather [Lim & Stolcke 91], and Trellis [Kilian 88]. Customization is often used in compilers that do not perform class hierarchy analysis, in order to obtain some information about the class of the receiver of a message. Profile-guided selective specialization, described in Section 4.2, is an alternative approach that uses class hierarchy analysis and profile data to determine where additional information is useful. In this section, we compare the two approaches, to examine their effects on performance and compiled code space. To do this, we examined the following compiler configurations, both with and without profile-guided class prediction:

- **intra** (or **i**) - The intra configuration from Section 5.2: 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 described in Section 5.2, which performs intraprocedural class analysis, class hierarchy analysis, and exhaustive class testing
- **i+CHA+exh+spec** - The same configuration described in Section 5.2, which performs intraprocedural class analysis, class hierarchy analysis, and exhaustive class testing, and selectively specializes routines

* These experiments were performed without exhaustive class testing, since we do not have a version of Vortex both that includes the exhaustive class testing algorithm and is capable of compiling the older version of the Cecil language that appears in some of these snapshots of the compiler benchmark.

based on profile data and a static analysis of the costs and benefits of specialization. Class hierarchy analysis is included because the selective specialization algorithm requires it.

To evaluate the performance of the algorithm, we measured both the dynamic number of dynamic dispatches in each version of the programs as well as the bottom-line execution speed and compiled code space. The results are shown in Table 10.

Table 10: Impact of specialization and customization

Program		Configuration	Execution time	Message sends executed	Compiled code space
Richards		intra	2.03 (1.00)	34,089,777 (1.00)	999,072 (1.00)
		i+cust	0.49 (0.24)	3,199,783 (0.09)	1,632,344 (1.63)
		i+CHA+exh	0.48 (0.24)	1,294,143 (0.04)	973,744 (0.97)
		i+CHA+exh+spec	0.45 (0.22)	1,345,337 (0.04)	982,056 (0.98)
	W/Profile	intra	0.43 (0.21)	5,313,897 (0.16)	1,024,504 (1.03)
		i+cust	0.47 (0.23)	3,199,884 (0.09)	1,622,904 (1.62)
		i+CHA+exh	0.35 (0.17)	723,861 (0.02)	981,888 (0.98)
		i+CHA+exh+spec	0.32 (0.16)	775,055 (0.02)	990,200 (0.99)
Deltablue		intra	16.46 (1.00)	199,620,412 (1.00)	1,017,888 (1.00)
		i+cust	7.55 (0.46)	44,854,060 (0.22)	1,759,184 (1.73)
		i+CHA+exh	8.69 (0.53)	49,182,401 (0.25)	1,009,408 (0.99)
		i+CHA+exh+spec	6.89 (0.42)	38,433,971 (0.19)	1,009,568 (0.99)
	W/Profile	intra	1.52 (0.09)	5,405,005 (0.03)	1,099,128 (1.08)
		i+cust	4.24 (0.26)	12,466,789 (0.06)	1,946,056 (1.91)
		i+CHA+exh	1.58 (0.10)	931,811 (0.00)	1,053,288 (1.03)
		i+CHA+exh+spec	1.56 (0.09)	931,811 (0.00)	1,053,400 (1.03)
Scheduler		intra	9.69 (1.00)	5,662,957 (1.00)	1,936,032 (1.00)
		i+cust	5.26 (0.54)	1,566,193 (0.28)	4,378,376 (2.26)
		i+CHA+exh	5.32 (0.55)	1,599,429 (0.28)	1,956,088 (1.01)
		i+CHA+exh+spec	4.98 (0.51)	1,401,864 (0.25)	2,029,040 (1.05)
	W/Profile	intra	3.11 (0.32)	466,123 (0.08)	2,047,688 (1.06)
		i+cust	3.41 (0.35)	593,122 (0.10)	4,745,104 (2.45)
		i+CHA+exh	2.74 (0.28)	377,142 (0.07)	2,008,728 (1.04)
		i+CHA+exh+spec	2.76 (0.28)	361,968 (0.06)	2,081,200 (1.07)

Table 10: Impact of specialization and customization

Program		Configuration	Execution time	Message sends executed	Compiled code space
Typechecker		intra	103.44 (1.00)	62,357,301 (1.00)	4,519,096 (1.00)
		i+cust	57.15 (0.55)	16,754,599 (0.27)	10,840,416 (2.40)
		i+CHA+exh	50.55 (0.49)	13,712,846 (0.22)	4,440,144 (0.98)
		i+CHA+exh+spec	46.49 (0.45)	11,031,174 (0.18)	4,594,656 (1.02)
	W/Profile	intra	33.84 (0.33)	6,961,025 (0.11)	5,090,344 (1.13)
		i+cust	37.12 (0.36)	6,885,251 (0.11)	11,918,720 (2.64)
		i+CHA+exh	29.78 (0.29)	5,605,325 (0.09)	4,735,416 (1.05)
		i+CHA+exh+spec	29.87 (0.29)	5,521,203 (0.09)	4,879,536 (1.08)
Compiler		intra	1,500.00 (1.00)	617,004,841 (1.00)	9,754,032 (1.00)
		i+cust	772.00 (0.51)	169,298,302 (0.27)	31,119,400 (3.19)
		i+CHA+exh	700.00 (0.47)	153,558,538 (0.25)	9,985,384 (1.02)
		i+CHA+exh+spec	639.00 (0.43)	129,811,734 (0.21)	10,302,752 (1.06)
	W/Profile	intra	615.00 (0.41)	124,908,520 (0.20)	10,708,512 (1.10)
		i+cust	532.00 (0.35)	79,130,611 (0.13)	31,200,736 (3.20)
		i+CHA+exh	486.00 (0.32)	71,685,978 (0.12)	10,343,920 (1.06)
		i+CHA+exh+spec	506.00 (0.34)	71,587,088 (0.12)	10,596,760 (1.09)

Both customization and selective specialization significantly decrease the number of messages sent by these programs, compared to the intra configuration, with selective specialization usually performing slightly better in terms of execution time and message sends executed. This trend is true both with and without profile-guided class prediction. However, these improvements over the intra configuration come at a cost. The cost for customization is paid in compiled code space and compile time: the executables for programs compiled with customization ranged from 60% to 220% larger, and compilation times increased by roughly the same factor. This cost also is higher for larger programs with more classes, deeper inheritance hierarchies, and more methods, making customization increasingly impractical for realistic sized programs. In contrast, class hierarchy analysis coupled with selective specialization provides better performance than customization and increases code space by less than 10% over the intra configuration. The costs of this approach are requiring that the entire class hierarchy of the program be available at compile time, precluding separate compilation, and gathering the profile data needed to derive the weighted call graph.

6 Related Work

6.1 Self

The Self-91 and Self-93 systems pioneered a number of techniques included in the Vortex compiler: Self-91 introduced intraprocedural static class analysis, splitting, customization, and fine-grained recompilation dependencies, and Self-93 introduced profile-guided receiver class prediction. The Vortex compiler extends the Self systems with class hierarchy analysis, exhaustive class testing and cone tests, a more powerful profile-guided specialization algorithm than customization, support for multi-method dispatching, and a more sophisticated selective recompilation mechanism. Vortex also is more language-independent, with C++, Modula-3, and Java front-ends nearing completion as of this writing.

The Self systems are based on dynamic compilation: no code is compiled until it is first invoked at run-time. Moreover, code is initially compiled with little or no optimization, and only if it is heavily executed is code dynamically recompiled with optimization. Dynamic compilation is exploited to good effect in several ways in the Self systems: it serves to eliminate compilation of unused code and even unused paths within procedures, it enables simple customization to be practical in many contexts, and it enables receiver class profile data to be gathered on-line as an application runs and then used when recompiling heavily-executed procedures with optimization. The Vortex system is based on more traditional off-line static compilation, in part so that applications can be compiled and delivered without requiring a compiler and some program representation to be present in the application's run-time system and in part to make it feasible to explore more expensive static analyses of programs, such as interprocedural analyses. It seems possible that a combined approach could work best: static off-line analysis of the program could lead to partially-compiled programs, with some parts of compilation deferred until run-time when more information becomes available. Several systems combining static analysis and dynamic code generation are being developed [Consel & Noel 96, Leone & Lee 96, Auslander et al. 96, Engler et al. 96].

6.2 Static Class Analysis

The general approach of class analysis based on sets of concrete classes started with the Typed Smalltalk system [Johnson 86, Johnson 88]. Most subsequent work on compiling object-oriented languages has used a similar framework. Palsberg and Schwartzbach used a type system based on "points" (single classes) and "cones" (all classes inheriting from some class) in their work [Palsberg & Schwartzbach 94].

A number of researchers have investigated interprocedural static class analysis. The key problem is that the call graph over which the interprocedural analysis should be run depends on the solution to the class sets inferred for the receivers of message send sites. Consequently, the call graph construction phase must be run in parallel with using the call graph to compute class sets. Another issue is how to cope effectively with both parametric and subtype polymorphism in programs; some amount of context-sensitive analysis is needed to avoid smearing together the distinct class information from different calling contexts. Palsberg & Schwartzbach developed an early algorithm characterized as solving a system of constraints [Palsberg & Schwartzbach 91], and later work with Oxhoj [Oxhøj et al. 92] extended this algorithm to handle polymorphic data structures better by treating each static occurrence of an object creation expression as producing a distinct type. Agesen adapted and extended this work to handle the Self language and to support better context-sensitivity [Agesen et al. 93, Agesen 95]. Plevyak and Chien have developed a sophisticated algorithm that works well for both polymorphic procedures and polymorphic data structures, based on iteratively solving the interprocedural analysis problem with greater degrees of context where needed, until the solution does not improve; their system also drives a procedure specialization algorithm that is based on the context-based specialization used during analysis. They have demonstrated very good results on small benchmarks, where nearly all run-time class tests and dispatches are removed. However, none of these algorithms have yet been shown to scale to programs over a few thousand lines; we have determined that a simple context-insensitive algorithm takes several hours to analyze our larger programs [Grove 95]. Additionally, issues such as separate or selective recompilation and incremental update of analysis results have not been addressed. Related interprocedural analysis problems arise in functional languages, where the function being called can be a computed expression, and a number of context-insensitive and -sensitive algorithms have been developed [Shivers 88, Heintze 94]. Similarly, algorithms have been developed to support calls of function variables in Fortran [Callahan et al. 90] and other imperative languages [Ryder 79, Weihl 80, Landi & Ryder 92].

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.

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.

Optimizing implementations of dynamically-typed languages like Lisp and Scheme often use techniques similar to static class analysis to try to eliminate the cost of run-time type checking [Kranz et al. 86]. Usually, these systems are simple intraprocedural dataflow analyses, with a fixed set of possible types derived from compiler knowledge about the type signatures of the primitive operations in the language. Some more sophisticated systems, such as Soft Typing and the Soft Scheme system [Cartwright & Fagan 91, Wright & Cartwright 94], support polymorphic interprocedural type inference. Soft typing attempts to use a modified version of Hindley-Milner type inference to perform type inference for Scheme, inserting run-time type checks and producing warning messages for the programmer wherever static inference could not prove the absence of run-time type errors. Static class analysis, being based on dataflow analysis, can generate flow-sensitive type bindings, while the Soft Scheme system accepts the normal type inference constraint of assigning a variable a single type throughout its lifetime, except in the special case of pattern-matching. Soft Scheme currently requires the whole program in order to perform its type inference, and it does not support incremental reanalysis.

The TIL compiler for ML uses "intentional polymorphism" to transform a reference to polymorphic routines into a reference to one of several overloaded monomorphic specializations, indexed by a run-time typecase [Tarditi et al. 96]; previous work by Leroy [Leroy 92] and Shao and Appel [Shao & Appel 95] has similar goals. TIL's propagation of concrete monomorphic type information in place of the more general (and less efficient) polymorphic routines is much like static class analysis, where concrete sets of representations are propagated. The TIL work introduces the typecases in place of references to polymorphic routines and then uses analysis to try to simplify the typecases where possible, while static class analysis introduces explicit typecases lazily when needed. TIL's analysis relies heavily on the static types declared or inferred for the ML program, while static class analysis can work for dynamically-typed languages; static type declarations can be included in the static class analysis in the form of cone representation constraints on variables. TIL specializes polymorphic functions for particular monomorphic argument types using the interprocedural type analysis mechanisms, while Vortex uses a separate profile-guided specialization mechanism. One of the key benefits of TIL's conversion of polymorphic code into monomorphic code is that basic datatypes can usually be represented in native, unboxed form without any tag bits or header words, making some basic operations more efficient, reducing the size and allocation overhead of objects, and easing interoperability across languages. These optimizations would be useful to adapt to the object-oriented context. At present, TIL has only been tested on relatively small ML programs without modules, where it is feasible to compile away all the polymorphism in the applications. It remains to be seen how well the approach will scale to programs of tens of thousands of lines.

Haskell implementations typically pass run-time dictionary data structures to implement manipulating values within a function polymorphic over all instances of a particular type class [Peterson & Jones 93]. The dictionary is similar to the run-time class associated with objects in an object-oriented system. Accordingly, static class analyses could be used in this context to reduce the overhead of accessing these data structures. Jones reports on a version of Haskell where he applies exhaustive procedure specialization to replace a single polymorphic function with a set of monomorphic versions, one for each calling tuple of dictionaries [Jones 94]. This is quite similar in flavor to the TIL compiler's replacement of polymorphism with specialized monomorphic versions, and customization in Self and similar systems. For programs of up to 15,000 lines, exhaustive specialization of polymorphic functions actually reduces code size.

6.3 Profile-Guided Class Prediction

Smalltalk-80 [Deutsch & Schiffman 84], Self-89 [Chambers & Ungar 89], and Self-91 [Chambers 92] compilers have all utilized hard-wired receiver class prediction (called *type prediction* in the Self work) to eliminate much of the overhead due to their pure object model and user-defined control structures. The Self-93 [Hölzle & Ungar 94b] work demonstrated that on-line call-site-specific profile-based class prediction (called *type feedback* in that work) can substantially improve the performance of Self programs, but it did not investigate the effectiveness of off-line profiling, demonstrate the applicability to hybrid languages such as C++, or examine the peakedness or stability of class distributions.

Calder and Grunwald considered several ways of optimizing dynamically-bound calls in C++ [Calder & Grunwald 94]. They examined some characteristics of the class distributions of several C++ programs and found that although the potential polymorphism was high, the distributions seen at individual call sites were strongly peaked. Our C++ results confirm this general result, but the programs we measured are significantly larger and appear to make heavier use of object-oriented programming techniques, such as deep class hierarchies and factoring, judging by the greater degree of observed polymorphism.

Wall [Wall 91] investigated how well profiles predict relatively fine-grained properties such as the execution frequencies of call sites, basic blocks, and references to global variables. He reported that profiles from actual runs of a program are better indicators than static estimates but still are often inaccurate. We investigated the predictive power of a coarser-grained kind of information, receiver class distributions, and found that they have much greater stability. We also examined stability across program versions, a property not considered by Wall.

Agesen and Hölzle compared the effectiveness of profile-guided receiver class prediction to interprocedural class analysis alone for a suite of small-to-medium sized Self programs [Agesen & Hölzle 95, Agesen & Hölzle 96]. The two techniques appeared roughly comparable in performance, and somewhat disappointingly there appeared to be no synergistic effects when combined together.

6.4 Procedure Specialization

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 optimized code; compile time is not reduced, however. 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). Lea describes a language extension to C++ to support customization on any argument [Lea 90]. Lea's approach didn't address selective specialization upon particular groups of classes.

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 the program's call graph, 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 complete call graph prior to analysis. Instead, we use a subset of the real call graph derived from dynamic profile information. This is important, because as discussed above when describing interprocedural class analysis, a precise call graph is difficult to compute in the presence of extensive dynamically dispatched messages. 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.

Procedure specialization has been long 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.

7 Conclusions

The Vortex compiler strives to make certain kinds of programming styles practical through a combination of static analyses and profile-guided optimizations, plus supporting compiler infrastructure. Programmers should feel free to develop reusable libraries and frameworks where "hooks" for later extension, in the form of dynamically-dispatched calls, can be used wherever sensible. This style can be followed in any object-oriented language, with enough programmer discipline or with a pure language that prevents premature binding to implementations. Since any given application using the library only exploits a subset of the available functionality and extensibility of the library, the compiler is responsible for eliminating the cost of unused flexibility and reducing the cost of the dispatching that is exploited. Cass hierarchy analysis works well to identify when subclassing and method overriding are not used by a particular combination of application and libraries, replacing unnecessary dynamic dispatching with direct calls or inlined code. Selective specialization applies code replication to turn shared polymorphic code into multiple versions of monomorphic code, thereby eliminating still more dispatches and reducing the performance cost of factoring; profile and class hierarchy information are used to focus specialization to the most important routines and argument type combinations. Profile-guided receiver class prediction reduces the cost of the remaining dynamic dispatching in the common case where one or a few receiver classes dominate. A selective recompilation mechanism helps make a whole-program approach to compilation and optimization practical in a normal program development setting, Performance for a group of Cecil benchmark programs ranging in size from a few hundred to 75,000 lines in size shows these techniques to together deliver nearly an order of magnitude improvement over unoptimized code, and a factor of four improvement over code compiled with static intraprocedural optimizations.

We are currently developing front-ends for C++, Modula-3, and Java for Vortex, to investigate how well our optimizations work for other points in the language design space [Dean et al. 96]. Since the programing style used is at least as important as the language the program is written in, we are developing language-independent metrics with which to characterize the style of the benchmark and compare that style to the performance improvements attributed to different optimization techniques.

We will be investigating a number of optimizations in the future. Effective, scalable interprocedural class analysis is a key technology, since the call graph it builds is required for several other interprocedural optimizations. We plan to investigate automatically specializing an object's layout, unboxing primitive datatypes, and eliminating pointer indirections to component objects, in a style similar to that in Shao and Appel's ML compiler [Shao & Appel 95] and in the TIL ML compiler [Tarditi et al. 96]; this optimization requires a call graph to track the flow of data structures through the program, in lieu of a static type system like ML's. Implementing abstract data structures using particular representations is useful for communicating with external programs and for writing systems programs. 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 exploit the program call graph. We also wish to investigate integrating static and dynamic compilation in the same system, combining the advantages of each and allowing further experimentation with different divisions of labor between static and dynamic work. Finally, we wish to explore different points along the spectrum ranging from pure separate compilation to pure whole-program optimization. For example, a 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 fulfils 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.

In high-level languages, aggressive and subtle optimizations can often make it difficult for programmers to predict program performance; the source code is no longer an acceptable model for relative costs of different versions of code. We want to investigate mechanisms for profiling and other performance monitoring to inform the programmer about different aspects of the program's behavior, ideally without requiring the programmer to have a detailed understanding of the compiler's analyses and optimizations. In addition, the programmer should be able to express assertions about their optimization expectations, for instance that some message and closure get inlined down into a simple loop, with feedback to the programmer whenever an expectation is not met.

The techniques developed in Vortex were originally aimed at object-oriented languages, but other high-level languages share many of the important characteristics of object-oriented languages, when viewed from the compiler's perspective. For example, generic arithmetic and other generic operations in Common Lisp and similar languages [Bobrow et al. 88] introduce overloaded operations over different run-time representations that are selected among at run-time, based on the run-time type of the operands; this is just a non-extensible kind of class hierarchy, and many of Vortex's techniques could be applied to optimizing this code. Recent ML and Haskell implementations [Jones 94, Shao & Appel 95, Tarditi et al. 96] are starting to optimize programs for particular representations in place of the original polymorphic version, and the analyses being used to track representations bear a great deal of similarity to static class analysis. Techniques in Vortex such as profile-guided optimizations and selective procedure specialization might be profitably added to these systems. Vortex's selective recompilation mechanism could help support the whole-program compilation model used in some of these systems.

Acknowledgments

Several other members of the Cecil and Vortex team have greatly contributed to our work. Vassily Litvinov gathered all of the performance data reported in Section 5. Charles Garrett performed much of the early work on gathering and studying the characteristics of receiver-class profiles. Tina Wong and MaryAnn Joy implemented initial versions of value-based analysis, eliminating redundant and dead loads and stores, and performing symbolic range analysis to eliminate unnecessary array bounds checks. Greg DeFouw implemented parts of the Vortex compiler's support for other input languages, particularly Modula-3.

This research was supported in part by an NSF Young Investigator Award (contract number CCR-9457767), an NSF Research Initiation Award (contract number CCR-9210990), a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM, Pure Software, and Edison Design Group.

References

[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. To appear.

[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, ACM SIGPLAN Notices*, pages 113–128, November 1991. Published as Proceedings OOPSLA '91, 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. To appear.

[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.

[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.

[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.

[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.

[Callahan et al. 90] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, 1990.

[Cartwright & Fagan 91] Robert Cartwright and Mike Fagan. Soft Typing. *SIGPLAN Notices*, 26(6):278–292, June 1991. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[Caseau 93] Yves Caseau. Efficient Handling of Multiple Inheritance Hierarchies. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 271–287, October 1993. Published as Proceedings OOPSLA '93, 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, ACM SIGPLAN Notices*, pages 1–15, November 1991. Published as Proceedings OOPSLA '91, 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.

[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, ACM SIGPLAN Notices*, pages 49–70, October 1989. Published as Proceedings OOPSLA '89, 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*, pages 221–230, Seattle, WA, April 1995.

[Chang et al. 92] Pohua P. Chang, Scott A. Mahlke, , and Willam Y. Chen 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 & Noel 96] Charles Consel and Francois Noel. A General Approach for Run-Time Specialization and its Application to C. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, Florida, January 1996.

[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.

[Cousot & Cousot 77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[Cytron et al. 89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 1989.

[Dean & Chambers 94] Jeffrey Dean and Craig Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 273–282, Orlando, FL, June 1994.

[Dean et al. 95] 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. 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. To appear.

[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:pldi Beyond k-Limiting. *SIGPLAN Notices*, 29(6):To Appear, 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. To appear.

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

[Engler et al. 96] Dawson R. Engler, Wilson C. Whsieh, and M. Frans Kaashoek. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, Florida, January 1996.

[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*.

[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.

[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 CAS-*

CON '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.

[Heintze 94] Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 306–317, Orlando, FL, June 1994.

[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 94a] 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 & Ungar 94b] 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 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.

[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.

[Johnson 86] Ralph E. Johnson. Type-Checking Smalltalk. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 315–321, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[Johnson 88] Ralph Johnson. TS: AN Optimizing Compiler for Smalltalk. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, pages 18–26, November 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[Jones 94] Mark Jones. Dictionary-Free Overloading by Partial Evaluation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation '94*, pages 107–118, Orlando, FL, June 1994.

[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.

[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.

[Knoop et al. 94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial Dead Code Elimination. *SIGPLAN Notices*, 29(6):147–158, June 1994. In *Proceedings of the ACM SIGPLAN '94 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.

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

[Leone & Lee 96] Mark Leone and Peter Lee. Optimizing ML with Run-Time Code Generation. *SIGPLAN Notices*, pages 137–148, May 1996. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[Leroy 92] Xavier Leroy. Unboxed Objects and Polymorphic Typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albequerque, New Mexico, January 1992.

[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.

[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*.

[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, ACM SIGPLAN Notices*, pages 146–161, November 1991. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

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

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

[Peterson & Jones 93] John Peterson and Mark Jones. Implementing Type Classes. *SIGPLAN Notices*, 28(6):227–236, June 1993. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[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.

[Ruf & Weise 91] E. Ruf and D. 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.

[Ryder 79] Barbara Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 5(3):216–225, 1979.

[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, ACM SIGPLAN Notices*, pages 9–16, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[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 Imple-*

mentation.

[Shivers 91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[SRC] DEC SRC Modula-3 Implementation. Digital Equipment Corporation Systems Research Center. http://www.research.digital.com/SRC/modula-3/html/home.html.

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

[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*.

[Tichy 85] Walter F. Tichy. RCS-A System for Version Control. *Software Practice and Experience*, 15(7):637–654, July 1985.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, pages 227–242, December 1987. Published as Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, number 12.

[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*.

[Weihl 80] William E. Weihl. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, Nevada, January 1980.

[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*.

[Wright & Cartwright 94] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 250–262, Orlando, FL, June 1994.