

# Vortex: An Optimizing Compiler for Object-Oriented Languages

Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers

Department of Computer Science and Engineering  
University of Washington  
Box 352350, Seattle, Washington 98195-2350 USA  
{jdean, gdefouw, grove, vass, chambers}@cs.washington.edu

## Abstract

Previously, techniques such as class hierarchy analysis and profile-guided receiver class prediction have been demonstrated to greatly improve the performance of applications written in pure object-oriented languages, but the degree to which these results are transferable to applications written in hybrid languages has been unclear. In part to answer this question, we have developed the Vortex compiler infrastructure, a language-independent optimizing compiler for object-oriented languages, with front-ends for Cecil, C++, Java, and Modula-3. In this paper, we describe the Vortex compiler's intermediate language, internal structure, and optimization suite, and then we report the results of experiments assessing the effectiveness of different combinations of optimizations on sizable applications across these four languages. We characterize the benchmark programs in terms of a collection of static and dynamic metrics, intended to quantify aspects of the "object-orientedness" of a program.

## 1 Introduction

In recent years, it has been demonstrated that intra- and interprocedural static class analysis [Chambers & Ungar 90, Plevyak & Chien 94, Agesen & Hölzle 95], class hierarchy analysis [Dean et al. 95b], and profile-guided receiver class prediction [Hölzle & Ungar 94, Grove et al. 95] can greatly improve the performance of dynamically-typed, purely object-oriented languages such as Cecil [Chambers 92, Chambers 93], Self [Ungar & Smith 87], and Concurrent Aggregates [Chien 93]. These techniques have been highly effective in this context, since message sends are ubiquitous and expensive; even the most primitive operations in these languages are implemented via user defined methods and dynamic dispatching. However, in statically-typed, hybrid object-oriented languages such as C++ [Stroustrup 91] and

Modula-3 [Nelson 91], much of the normal execution of programs involves non-object-oriented constructs, and consequently the incidence of dynamically-dispatched calls is much lower than in pure languages. Thus, is it unclear how much overhead due to use of object-oriented features exists in programs written in these languages, and it is unclear how much benefit can be gained by applying advanced optimizations of the object-oriented features to programs in these languages. Java [Gosling et al. 96], a language somewhere between C++ and Cecil in terms of its purity and dependence on object-oriented features, offers another interesting point in the language design space, and therefore its need for advanced optimization techniques may be different than both Cecil and C++.

In this paper, we present a study of the effectiveness of advanced object-oriented-focused optimizations across a range of object-oriented languages (C++, Modula-3, Java, and Cecil). The study was conducted using the Vortex compiler, a language-independent optimizing compiler for object-oriented languages that we have designed and implemented. By compiling all programs in these languages with a common optimizing back-end, we can ensure that each program receives comparable treatment and optimization effort.

The next section of this paper reviews the optimization techniques included in Vortex. Section 3 describes the structure of the Vortex compiler, focusing on the design of its intermediate language and reporting on our experience in building an optimizing compiler for a wide range of languages. In Section 4 we describe the benchmark programs used in this study. As part of this description, we define a collection of static and dynamic metrics with which to characterize the programs, and we use these metrics both to better convey the internal structure of the programs and also to attempt to predict when different optimization techniques are likely to be most effective in improving an application's performance. Section 5 presents the performance results. Section 6 describes related work, and we offer our conclusions in Section 7.

## 2 Background

A compiler can replace a dynamic dispatch with a static call whenever it can determine that a single method will be invoked for all possible receiver classes of that call site. A message send that is replaced by a call in this fashion has been *statically bound*. Vortex uses five main techniques, described in the following subsections, to statically bind message sends. Without additional programming environment support, interprocedural optimizations like class hierarchy analysis and cross-module inlining preclude rapid turnaround after incremental programming changes. Section 2.5 briefly describes the selective recompilation mechanism provided by Vortex to allow whole-program optimizations and day-to-day application development to coexist.

### 2.1 Intraprocedural Class Analysis

Intraprocedural class analysis uses a standard iterative dataflow approach to compute for each program expression a set of classes such that any runtime value of the expression is guaranteed to be an instance of one of the classes in the computed set [Johnson 88, Chambers & Ungar 90]. The analysis maintains a mapping from variables to sets of classes, and propagates this mapping through the procedure's control flow graph. By default, variables map to the set of all possible classes, but literals and the results of object allocations (*new*) are mapped to singleton class sets. Class sets are combined with set-union at control flow merge points, and class sets are narrowed after a run-time class test conditional branch.

When a dynamically-dispatched message send is encountered, this mapping is consulted to determine if the receiver of the message is known to be a single class, or a bounded union of classes. If only a single receiver class is possible, or if all classes in a union invoke the same method, then the message send can be statically bound. Because intraprocedural class analysis has only local knowledge, it cannot statically bind a message send if the receiver is known only to be an instance of class *C* or some subclass of *C*, since an unknown subclass of *C* may provide an overriding definition of the target method. This limitation implies that knowing the static type of the receiver of a dynamically-dispatched message, as is the case in statically-typed languages like C++, Modula-3, and Java, is insufficient on its own to enable intraprocedural class analysis to statically bind the message send, since the possibility of an overriding method defined on a subclass cannot be ruled out.

### 2.2 Class Hierarchy Analysis

Class hierarchy analysis [Dean et al. 95b] broadens the scope of the information available to the compiler by giving it access to all of the class and method declarations in the program. Given this global knowledge of the class hierarchy, the

compiler can convert unbounded information of the form “a variable holds an instance of some (unknown) subclass of *C*” into a bounded set of possible classes. Only bounded sets of classes can provide useful information to the optimizer. Thus, class hierarchy analysis addresses one of the key weakness of intraprocedural class analysis by enabling the conversion of unbounded sets of classes, derived from static type declarations or from the method's specialized formal parameters, into bounded sets of classes.

### 2.3 Receiver Class Prediction

There are message sends that cannot be statically bound solely through static analysis; some message sends *do* invoke more than one method at run-time. However, it is still possible to transform message sends of this type into a form that allows inlining of at least a subset of the possible target methods. Receiver class prediction is a simple local code transformation that converts a dynamic dispatch into a run-time type-case structure: one or more explicit in-line tests for particular expected classes, each of which branches when successful to a statically-bound or inlined version of the target method for that class, possibly followed by a final dynamic dispatch to handle any remaining unpredicted classes. Receiver class prediction can be driven either by information hard-wired into the compiler, as in early Smalltalk and Self implementations [Deutsch & Schiffman 84, Chambers & Ungar 89], or by profile-derived class distributions [Hölzle & Ungar 94, Grove et al. 95], or by static examination of the program's class hierarchy [Chambers et al. 96]. We use the term *exhaustive class testing* to refer to class-hierarchy-guided class testing, and *profile-guided class prediction* to refer to class testing based on dynamic profile information.

If a large number of receiver classes are possible at a call site, testing for individual classes can be very expensive. However, if the number of target methods is low, then run-time *subclass* tests can be inserted instead of class identity tests, leading to a number of run-time tests on the order of the number of possible methods rather than the number of possible classes.

It is quite common for a method to contain multiple message sends to a single receiver value; for example, several messages might be sent to *self*. If receiver class prediction is applied to each of these message sends, redundant class tests will be introduced. Splitting can eliminate these redundant tests by duplicating paths through the control flow graph starting at merges after one test and ending with the redundant test to be eliminated [Chambers & Ungar 90].

### 2.4 Customization and Specialization

Customization and method specialization [Chambers & Ungar 89, Lea 90, Dean et al. 95a] are techniques that also can be used to statically bind messages sent to *self*, by creating

multiple compiled copies of a single source method, each specialized to particular receiver classes. However, since Vortex does not support customization or specialization for languages that use dispatch tables to implement message sends (e.g., C++ and Modula-3), we do not consider this technique in our study.

## 2.5 Selective Recompilation

Because of their global scope, interprocedural optimizations like class hierarchy analysis and cross-module inlining can introduce non-local dependencies in the compiled code, thus preventing strict separate compilation. However, it is still possible to achieve rapid turnaround after programming changes if the implementation keeps track of these dependencies and implements selective recompilation. The Vortex compiler records the non-local effects of whole-program optimization in a dependency graph, which is stored in a persistent program database. When pieces of the source program are modified, the dependency graph is consulted to determine which object files must be recompiled. Our experience with Vortex has been that incremental compilation is quite practical, and that most programming changes result in very few additional files being recompiled that were not directly modified [Chambers et al. 95].

## 3 Vortex Compiler Infrastructure

The basic structure of the compiler is shown in Figure 1. Each of the different front-ends does whatever parsing and typechecking are appropriate for its input language, and then translates the input into the Vortex compiler's intermediate language (IL), which is described in Section 3.1. The IL representation of the program then proceeds through the various stages of Vortex's optimizer until code is generated; this process is outlined in Section 3.2.

The usual way of adding a new source language to the Vortex back-end is to reuse an existing public-domain or commercial front-end, and modify it to output the Vortex IL. For C++, we started with the Edison Design Group's C++ Front End [EDG], a commercial product used as the front-end for many industrial C++ compilers. For Modula-3, we modified the front-end from the freely-available DEC SRC Modula-3 implementation [SRC]. For Java, we first use Sun's `javac` Java compiler program [JDK] (invoked with the `-O` flag to perform as much optimization as possible in the existing front-end, to avoid overstating the benefits of Vortex's optimizations) to translate Java source programs into Java bytecodes, and then we modified the `javap` bytecode "disassembler" to convert to Vortex IL. Since no public-domain or commercial Cecil front-ends were available, we developed one from scratch.

## 3.1 The Vortex Intermediate Language

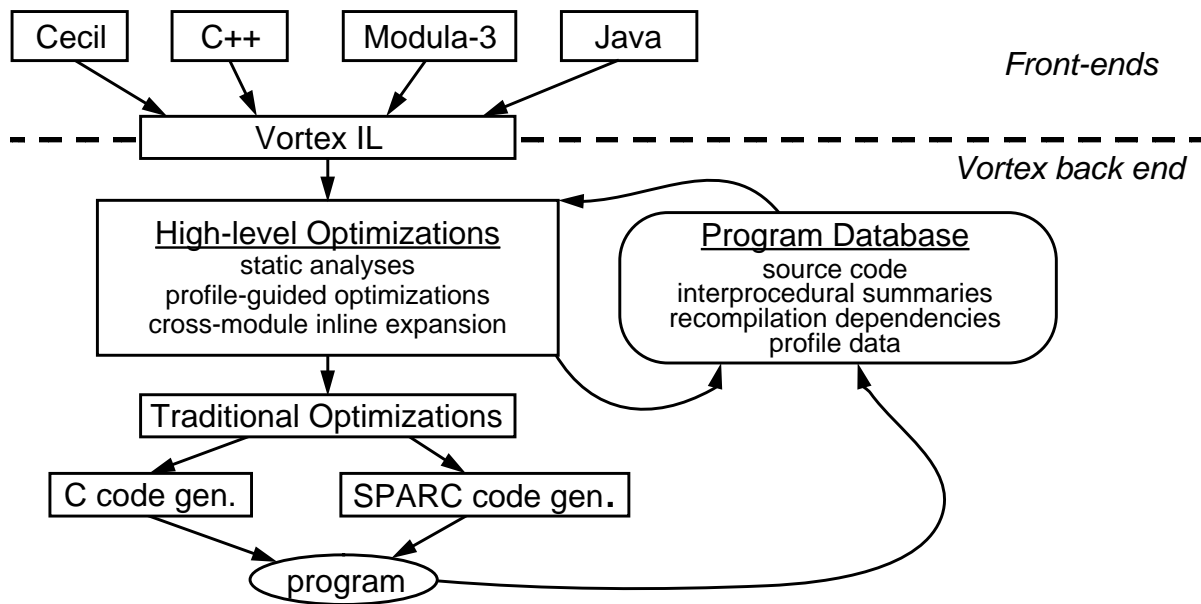
The Vortex intermediate language is the interface between the different language front-ends and the compiler back-end, and as such, it must allow the front-ends to describe the structure of a program in such a way that the optimizations described in Section 2 can be performed. In particular, many existing compilers for object-oriented languages (such as the DEC SRC Modula-3 implementation) translate input programs into a generic low-level form, where the object-oriented features have been translated into sequences of standard imperative constructs. This enables traditional back-ends to cope with the new features, but it blocks analyses and optimizations that need higher-level information about the original language constructs. The Vortex intermediate language supports both traditional lower-level operations and selected higher-level operations making things such as message sends, field accesses, run-time type tests, and object creations explicit. Individual language front-ends can choose high-level or low-level translations for their language features. As Vortex optimization phases proceed, higher-level constructs are successively expanded into lower-level constructs, until code generation takes place. The rest of this section describes the primary global declarations and the kinds of executable statements supported by the Vortex IL.

### 3.1.1 Classes and the Inheritance Hierarchy

Vortex represents the program's classes as a directed acyclic graph. The inheritance structure of the program is communicated to the back-end by declaring each class and the (possibly empty) list of superclasses from which it inherits. The back-end supports multiple, non-replicating inheritance (virtual inheritance in C++ terms). We currently rely on the language front-end to handle other forms of inheritance, such as replicating (non-virtual) inheritance in C++, although the back-end could be extended to support other models of inheritance as well. A sample class declaration in the Vortex IL is:

```
class Square isa Rhombus, Quadrilateral;
```

To implement lookup rules for operations such as method lookup and instance variable lookup, Vortex treats the inheritance graph as a partial order, choosing the most specific match according to the partial order and reporting an ambiguity otherwise. This partial ordering implements the simple rule that children override parents, which was derived from the Cecil language semantics [Chambers 93]. This can accommodate a wide range of languages whose semantics either match these or that have static typechecking rules that are more restrictive. For example, C++ has a stricter treatment of ambiguities, but C++'s static typechecker ensures that no programs with ambiguities according to C++'s rules produce IL code. Vortex's semantics are sufficient to handle the



**Figure 1:** Vortex Architecture

inheritance rules for single-inheritance languages such as Java, Modula-3, and Smalltalk [Goldberg & Robson 83], and to handle multiple-inheritance languages such as Cecil, C++, and Trellis [Schaffert et al. 85]. However, some languages use other rules for method lookup (such as CLOS, which uses a left-to-right ordering of parents for resolving ambiguities [Bobrow et al. 88]), and compiling such languages would require adding back-end support for more generally computing the required partial order from the inheritance graph.

C++ and Modula-3 support parameterized classes and/or methods (templates in C++ and generic modules in Modula-3). Currently, the front-ends for these languages expand away the templates, so Vortex sees only unparameterized classes and methods. This strategy is in keeping with how other compilers implement these languages, but it does prevent Vortex from performing optimizations to share code across template instantiations where possible.

### 3.1.2 Instance Variables and Representations

Classes can have many different possible representations, depending on how their instance variables are laid out and other factors such as alignment, padding, and the sizes of numeric data types. Representation decisions must be made at some level, and Vortex supports two strategies: either the front-end can lay out classes and structures completely, or they can provide information about instance variables to the back-end and allow the back-end to make layout decisions. Each approach has advantages and disadvantages. Having the front-end perform class layout, which is the approach we have

chosen for the C++ and Modula-3 front-ends, preserves compatibility with separately-compiled code that uses the same layout rules and avoids having to encode class layout rules for these languages in the back-end. However, having the back-end perform class layout, which is the approach we have chosen for the Cecil and Java front-ends, supports higher-level analysis of object structures (such as better alias analysis) and opens the opportunity for optimizations that rearrange data representations. (Recent work has shown that these can have substantial performance impact in the context of both Modula-3 [Fernandez 95] and ML [Shao & Appel 95, Tarditi et al. 96].)

If the front-end makes representation decisions, objects are described as a generic array of bytes of a particular size. Low-level pointer operations are used to implement instance variable accesses, and object allocation is implemented in terms of `malloc`-type byte allocators. On the other hand, if the back-end makes representation decisions, then the front-end declares the instance variables of a class, including their representations, and instance variable accesses and object allocations are represented directly in the IL. Array accesses similarly have both low-level (pointer-based) and high-level (direct) forms.

Some languages, including Cecil, C++, Java, and Trellis, allow an instance variable to be declared immutable, and the Vortex IL supports this annotation on instance variable declarations. Knowing an instance variable is immutable allows the optimizer to retain its knowledge about the instance variable's value, if any, across calls. For languages that use low-level accesses to instance variables, Vortex IL allows load

instructions to be marked as from immutable locations, enabling similar preservation of knowledge about that memory location. This low-level form works even if the memory location is only invariant in some contexts. For example, in C++, “vtbl” words of objects are unchanged outside of their constructor methods [Stroustrup 87].

The Vortex IL supports global variable declarations. A declaration specifies a representation and optionally the initial value for the variable (even if the variable is an aggregate data type).

### 3.1.3 Methods and Procedures

The Vortex IL uses a notion of a generic function [Bobrow et al. 88] to unify the concepts of procedures, methods, and multi-methods. Each call or message maps to a single generic function. A generic function contains a set of dynamically-overloaded methods, with each method indicating where in the program’s class hierarchy it is attached. A regular procedure is modeled as a singleton generic function whose sole method is attached to the (perhaps implicit) root of the inheritance graph, while several singly- and/or multiply-dispatched methods may be grouped together in one generic function. Static overloading as in C++ and Java is resolved by the front-end using a “name mangling” technique to encode the static argument types in the name of the message being sent; only dynamic overloading remains in the Vortex IL. Method lookup and inlining work uniformly over this generic function and method model.

A procedure or method implementation is communicated to the back-end via a method definition in the Vortex IL, which specifies the method’s name, a list of formal arguments and representations, a return representation, and a body of code made up of Vortex IL statements (see Section 3.1.4 below). To indicate where a method is placed in the inheritance hierarchy, a separate *associate* declaration is produced by the front-end that names the generic function and the tuple of argument class specializers for the method. Singly-dispatched methods will have non-any specializers only on the receiver argument; multi-methods may specialize any of their arguments. To illustrate how method and *associate* declarations fit together, consider the following C++ class:

```
class Square: Rhombus, Quadrilateral {
    virtual void draw(int color);
};
```

If *draw* were introduced in the *Shape* base class, the front-end might generate the following IL declarations:

```
class Square isa Rhombus, Quadrilateral;
method draw__6Square_int(this,color):void
{... IL code for draw...}
associate draw__6Square_int with
draw__5Shape_int(@Square, @any);
```

All methods that are part of the same generic function share the same message name in their *associate* declarations.

Initially, we had a single construct in the IL that subsumed both method and *associate* declarations. However, we changed this when we began work on the Modula-3 front-end, since Modula-3 allows arbitrary procedures to be used as methods, perhaps in multiple places. Also, giving each method a unique name separate from the generic function’s allows direct non-dispatched calls to methods to be represented easily.

### 3.1.4 IL Statements

Executable code in the Vortex IL is represented as a series of three-address statements [Aho et al. 86], including the usual complement of arithmetic, logical, pointer, branching (conditional, unconditional, and indexed), and direct and indirect procedure call-related statements. In addition, a number of higher-level object-oriented notions are reified in the Vortex IL, allowing the optimizer to more easily reason about them:

- **Message sends.** Message sends are a high-level operation in the Vortex IL that provide sufficient information for the back-end to perform optimizations like class hierarchy analysis and receiver-class prediction. As an example, consider the following C++ code:

```
Shape* s = ...;
s->draw(color);
```

This send of *draw* might be translated into the following Vortex IL statement:

```
send draw__5Shape_int(s, color);
```

Subsection 3.1.5 describes two strategies for how Vortex implements a *send* statement.

- **Instance variable accesses.** For languages where the front-end generates instance variable declarations and expects the Vortex back-end to perform object layout, accesses to instance variables are represented with instructions that specify the object being accessed, the name of instance variable, and the name of the class where the instance variable was declared (in case there are instance variables with the same name but declared in different classes). For example, consider the following Java code:

```
Rectangle r = ...;
r.upper_left = 100;
```

This instance variable access is translated into the following Vortex IL statement, assuming that the *upper\_left* instance variable was introduced in the *Rectangle* class:

```
r.upper_left@Rectangle := 100;
```

- **Object allocations.** A simple new statement is supported to allocate objects of a particular class. Static class analysis can determine the static type of the result of this statement exactly.
- **Run-time class tests.** Vortex IL supports both class identity tests and subclass tests as explicit comparisons. For example:

```
if s is_class Rectangle goto ...;
if s inherits_from Shape goto ...;
```

The Java front-end outputs these tests to implement dynamically-checked type casts. Although not currently implemented in high-level form, the Modula-3 front-end could also use these run-time tests for its TYPECASE and NARROW features.

Subsection 3.1.6 describes implementation strategies for class tests.

- **Type assertions.** Type assertions communicate static class information from the front-end to the back-end. In general, a type assertion is a guarantee from the front-end to the back-end that a particular value holds an instance of a particular set of classes at a particular program point. Similarly to run-time class tests, Vortex IL supports class identity and subclass assertions:

```
assert_type r is_class Rectangle;
assert_type s inherits_from Shape;
```

Class identity assertions are output by the front-ends when a language-level operation has just created a value that is known to be of a specific class (e.g., allocating a new object). Subclass assertions are output whenever the front-end has information about the static type of a value that might be useful to the back-end optimizer (e.g., the static type of the arguments of a method or the result of an instance variable access).

### 3.1.5 Implementing Message Sends

Vortex currently supports two implementation strategies for message sends, one based on dispatch tables and one based on polymorphic inline caches (PICs) [Hölzle et al. 91]. For Cecil and Java, we select the PIC-based implementation strategy: dynamically generate a piece of executable code per call site that linearly tests for the  $N$  most common receiver classes, falling back to a hash table lookup if the cache overflows. The PIC-based strategy is general, since it does not require dispatch table layout algorithms, but it can be less efficient than table-based implementations.

For C++ and Modula-3, we implement message sends using the language’s native table indexing strategy, and rely on the language front-end to generate the appropriate tables (i.e., virtual function tables in C++ and method suites in Modula-3). For the back-end to produce the right table indexing sequence

when implementing a send in lower-level terms, we augment the send IL statement to include the offset of the class identifier in the receiver object (e.g., the location of the “vtbl” pointer in the object in C++), as well as the entry number to use in the dispatch table. This information is sufficient to generate the appropriate code to call indirectly through the dispatch table.

Keeping the table generation in the front-ends has the advantage of not requiring the back-end to deal with the complexities of dispatch table layout (which is difficult to do in the presence of multiple inheritance), and it preserves compatibility so that we can call separately-compiled code from our generated code. However, it places some limitations on the optimizations that the back-end can perform. In particular, because dispatch tables are not exposed at a high enough level, the back-end cannot perform optimizations that would require generating different dispatch tables. Examples of such optimizations include customization and specialization [Chambers & Ungar 89, Lea 90, Dean et al. 95a] and converting a C++-style method from virtual to non-virtual to reduce the size of dispatch tables, when the back-end is able to detect that the method is not overridden anywhere in the program. However, in the presence of class hierarchy analysis, Vortex is able to streamline the virtual function calling sequence for C++: by detecting portions of the class hierarchy that do not use multiple inheritance, one pointer adjustment per call can be eliminated.\*

### 3.1.6 Implementing Run-Time Class Tests

For most languages, implementing a class identity test is straightforward: load the class identifier from the object (at a language-specific offset) and test it against a constant (derived from the class in a language-specific way). However, in C++ this test is complicated by the fact that, in the presence of multiple inheritance, an object can have multiple different “vtbl” addresses, each for different statically-typed views of the object [Stroustrup 87]. To support C++-style class identifiers effectively, the front-end annotates each class with a mapping from static type views to the name of the vtbl constant stored at that offset. The back-end then uses the known static type of the object being tested to determine which vtbl constant to compare against.

Efficient constant-time subclass testing can be implemented in languages having only single inheritance with a load instruction or two followed by a pair of comparisons. In languages with multiple inheritance, subclass testing is more

---

\* Trampoline functions are another technique for eliminating this pointer adjustment in the common (non-multiple-inheritance) case, but they incur a higher cost when multiple inheritance is actually used.

complex. In our Vortex implementation, we simply compute an  $N \times N$  boolean matrix of subclass relations, where  $N$  is the number of classes in the program. The  $\langle i, j \rangle$ -th entry indicates whether or not the class with the unique id  $i$  is a subclass of the class with id  $j$ . Alternative encodings of the subclassing relationship are possible, representing different time-space tradeoffs; Ait-Kaci *et al.* provide a useful overview of efficient lattice operations that discusses many of these alternatives [AK et al. 89].

### 3.1.7 Exceptions

Many languages support some form of exceptional control flow: Modula-3 supports traditional exceptions, C++ and Java support object-based exceptions, and Cecil, like Smalltalk and Self, has non-local returns that allow a nested closure to return directly from its lexically-enclosing method. The Vortex IL describes exceptions by treating calls and sends that can raise exceptions as control flow branches, with the exceptional return branch passing to an exception handler label. Vortex supports two implementation strategies for exceptions: one based on frame-by-frame propagation (using special return calling conventions and tests after each procedure call that can raise an exception), and one based on the language’s runtime system’s `setjmp/longjmp` mechanism (where a `setjmp` is performed when entering a “try” block and a `longjmp` is invoked to raise an exception). Cecil and Java use the propagation-based method, while Modula-3 uses its own `setjmp/longjmp`-based mechanism; none of our C++ programs use exceptions.

Originally, we used a lower-level form of exceptions for Modula-3, with the front-end generating explicit calls to `setjmp` to mark the beginning of TRY blocks. However, the Vortex optimizer was performing incorrect optimizations because its dataflow analysis did not understand the strange control flow that can be introduced in the presence of `setjmp`. To avoid sacrificing optimization in the presence of exceptions, which is the normal strategy, we opted to introduce a high-level explicit form for specifying the possible control flow in the presence of exceptions, and Vortex’s dataflow analyses are left unhindered. However, to safely compile programs that call `setjmp` explicitly, Vortex’s dataflow analyses should be modified to cope with its irregular control flow, under the assumption that explicit calls to `setjmp`, other than for implementing exceptions, are likely to be rare.

## 3.2 Optimization Phases

Vortex takes the IL representation of the program produced by one of its front-ends and performs a series of analyses and transformations on its way to generating optimized target code. As part of this process, high-level IL operations are

lowered by expanding them into sequences of simpler operations. In this section we highlight some of the interesting stages in this transformation.

A central piece of supporting infrastructure in this process is Vortex’s iterative dataflow analysis framework. All of the analyses and transformations rely on this framework to manage the details of iterative dataflow: control flow graph traversal, merging dataflow information at control flow merges, fixed-point convergence testing for loops, and graph transformations. Clients of the framework simply provide an analysis closure that encapsulates the appropriate flow functions and an analysis-specific domain class that implements the `copy`, `meet`, and `reached_convergence` methods. (Vortex’s IDFA framework is similar in spirit to the Sharlit system developed by Tjiang and Hennessy [Tjiang & Hennessy 92].) Analyses written separately using Vortex’s framework can also be composed so that they run together simultaneously, thus alleviating potential phase-ordering problems. (Click and Cooper describe the theoretical conditions under which the resulting combination of optimizations will produce better results than repeated applications of the original separate optimizations [Click & Cooper 95].)

The major phases in Vortex’s compilation of a control flow graph are:

- *Loop identification:* A dominator-based algorithm identifies the loops and the loop nesting structure of the procedure being compiled. (Vortex alternatively allows front-ends to provide this information directly, avoiding recomputing it in the back-end.) Vortex’s IDFA framework currently requires that control flow graphs be reducible (roughly, have a single entry node) [Aho et al. 86]; the loop identification pass detects irreducible flow graphs and reports them as errors.
- *Object-oriented optimizations:* This phase utilizes the iterative dataflow analysis framework’s composer interface to run a number of separately-written optimization passes simultaneously, thus potentially arriving at a better final fixed-point due to the synergistic relationships among the passes. Intraprocedural class analysis, class hierarchy analysis, profile-guided receiver class prediction, inlining, splitting, must-alias analysis, and an enhanced common subexpression elimination (CSE) pass all run in parallel. In addition to standard CSE transformations, the enhanced CSE pass also performs constant and copy propagation, constant folding, simplification of arithmetic operations, and elimination of redundant load and store operations. Because inlining is included as an integral part of the combined pass, when a routine is inlined it is immediately optimized. This

allows the callee to be fully optimized in the context of its caller and for the downstream code of the caller to benefit from any information gained by inlining the callee [Chambers & Ungar 90].

- *Closure optimizations*: Partial dead code elimination delays closure object creations until absolutely necessary (hopefully removing them entirely from the common case paths) and environments are marked to be either heap-allocated or stack-allocated. This phase has no effect in languages lacking closures or nested procedures.
- *Lowering I*: High-level operations like table sends and class tests are expanded into an equivalent sequence of lower level operations. All accesses to variables defined in lexically-enclosing scopes are expanded into an explicit series of loads.
- *Standard optimizations*: A suite of traditional optimizations such as CSE (again, since new opportunities for optimization have been exposed during the lowering phase), dead store elimination, and dead assignment elimination are applied.
- *C code generation*: If Vortex is generating C code, then code is produced now and compilation ends. The generated C code is portable across platforms with the same word size. Otherwise the following additional stages occur:
- *Lowering II*: Remaining high-level nodes and complex operators are expanded. After this second lowering phase, most IL operations can be implemented in a single machine instruction.
- *Standard optimizations repeated*.
- *Low-level optimizations*: Graph coloring-based intraprocedural register allocation and instruction scheduling.
- *Assembly code generation*.

### 3.3 Experience

Vortex originally was a Cecil-specific compiler, but over the last year or so we reworked it to be more language-independent. Starting with Cecil first had both benefits and drawbacks:

- The original Cecil compiler already supported a set of sophisticated optimizations aimed at a powerful set of language features, such as multiple dispatching, multiple inheritance, and closures. Since the other languages we compile mostly have subsets of Cecil's object-oriented language features, only a few small changes were needed for the Vortex compiler; `associate` declarations are one enhancement of the original Cecil compiler, and we had to generalize Cecil's exception handling support to cover what is used in Modula-3 and Java.

- The original Cecil compiler was geared towards a language with a type-safe, pure data model. For Vortex, we needed to do significant retrofitting to support non-word-sized datatypes, uncontrolled pointers, structures and arrays with non-scalar elements, and so on.

Both Modula-3 and Java had nicely-modularized existing front-ends that presented a good interface for compiling into the Vortex IL: DEC SRC Modula-3 supports a separate code-generator interface, which we added a new implementation of, and Java defines a bytecoded intermediate language that we translated in a straightforward way into the Vortex IL. DEC SRC Modula-3's code-generator interface includes only low-level operations, so we had to augment the interface with high-level operations; Java's bytecodes are already at the appropriate level for Vortex.

EDG's C++ front-end was also reasonably modular, although we had to start with an annotated parse tree and implement intermediate code generation to the Vortex IL. The complexity and size of the C++ language made this a more difficult task than the other two languages, but the real obstacle to gathering and compiling C++ applications with Vortex is that there is no well-defined C++ language in common use: each program we gathered would only compile on a subset of compilers, or used compiler-specific extensions, or used a version of the language that was different than what EDG's front-end expected. This situation makes it quite difficult to do compiler research for C++.

A final difficulty is that some C compilers cannot cope with the C code Vortex produces. We had to go to some lengths to produce C code that did not have functions or basic blocks that were too long or to produce files that had too many global symbols. Machine-generated programs have quite different characteristics than human-generated programs, and we encountered the same kinds of problems as other researchers in trying to compile our output.

## 4 Metrics for Describing Object-Oriented Programs

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



## 4.1 Metric Definitions

We considered a number of different metrics for characterizing our applications. After evaluating how well they captured the underlying program structure and usage of object-oriented language features, we selected the following metrics as being the most illuminating:

- *Number of Immediate Parents*: Measures the number of immediate parents of each class in the program; indicates the degree to which multiple inheritance is utilized.
- *Number of Immediate Children*: Measures the number of classes that directly inherit from each class in the program; indicates the branching factor (breadth) and “bushiness” of the class hierarchies.
- *Maximum Distance to Root of Inheritance Hierarchy*: The longest path from each class to the root of its inheritance hierarchy; indicates the depth of the class hierarchies used by the program.
- *Number of Applicable Methods*: Measures the number of applicable methods at a dynamically-dispatched call site, optionally weighted by the execution frequency of the call site. Class hierarchy analysis works well when a large number of call sites have a single applicable method, while exhaustive class testing applies when a few methods are possible at a call site.
- *Class Test Efficiency*: Measures the fraction of calls at a call site that go to the most common receiver class at that call site, weighted by the execution frequency of the call site. This histogram will always be a subset of the one representing the dynamic number of applicable methods at a call site. Profile-guided receiver class prediction can work well if the most common class at a call site is much more common than other classes; similarity between this histogram and the histogram for the dynamic number of applicable methods implies that dynamically-important call sites are dominated by a single receiver class.
- *Average Cycles Between Message Sends*: Measures the average number of cycles elapsed between message sends. As this value increases, we expect the overall performance impact of the optimizations to decrease.

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

95]. Due to space constraints, we only use the “first class same” metric here.

## 4.2 Applying the Metrics

We applied the metrics defined in the previous section to a number of medium-to-large applications written in Cecil, Java, Modula-3, and C++. Table 1 summarizes several distinguishing language features.

**Table 1: Language Characteristics**

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

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

Several of these language characteristics have a large impact on the effectiveness of the object-oriented optimizations. Because of the much higher frequency of message sends in pure languages compared to hybrid languages, the impact of the object-oriented optimizations is much more dramatic. Static type declarations are excellent fodder for class hierarchy analysis, so one would expect it to be more effective in statically-typed languages.

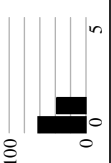
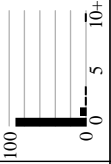
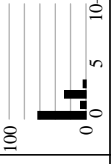
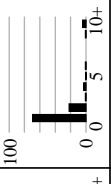
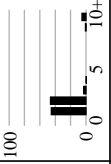
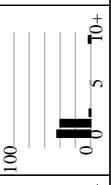

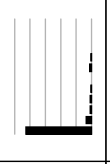

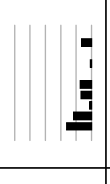
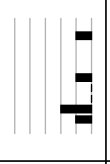
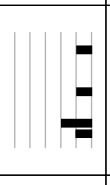
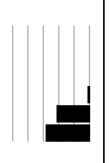
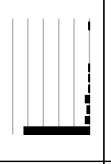
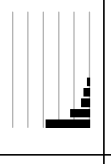

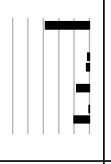
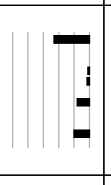
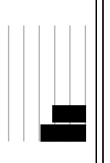
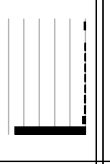
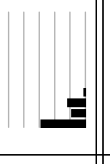

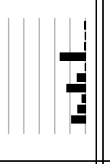
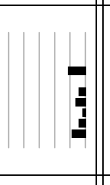
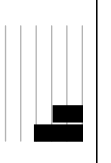
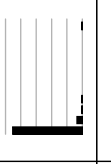
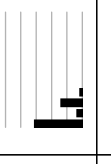

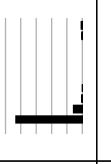
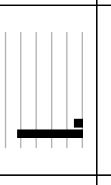
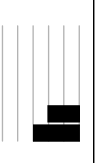
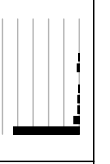

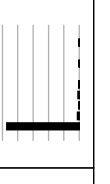
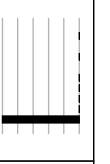
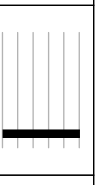
Table 2 describes the application suite and presents the results of applying the metrics to the programs. We use histograms to visually display the metrics; the height of each bar represents the percentage of all elements whose metric value corresponds to the bar’s *x*-coordinate. For C++, we also examined a version of two of the benchmarks where we hand-modified the programs to make all methods virtual, to explore an alternative programming style; these two benchmarks are identified with an `-av` suffix.

All of our benchmarks are substantial in size, with all at least 10,000 lines and most over 20,000. The Cecil programs have the largest and deepest class libraries, with `javac` and

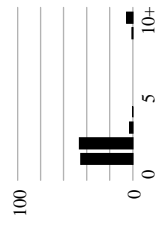
Table 2: Application Descriptions

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

Table 2: Application Descriptions

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

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



m2tom3 also having deep class hierarchies, but all the programs except **prover** have at least a hundred classes and 400 methods. Cecil programs used multiple inheritance a moderate amount, and the Java programs used multiple subtyping some, but the C++ programs made little if any use of multiple inheritance.

Examination of the static number of call sites with only one applicable method would suggest that class hierarchy analysis could be effective in most of the benchmarks, but the version weighted by dynamic execution frequency indicates a somewhat lower expected benefit; **eon** and **ktsim-av** buck this trend. As expected, the C++ all-virtual programs have much greater number of call sites amenable to class hierarchy analysis than the regular versions of these benchmarks. About half the benchmarks have significant numbers of messages with 2-3 applicable methods, indicating that exhaustive class testing could be beneficial (although Vortex does not support this technique for C++ or Modula-3 yet). Since the histograms reporting the efficiency of profile-guided class testing are usually similar to the dynamic number of applicable method histograms, profile-guided receiver class prediction looks promising.

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

## 5 Performance Studies

To assess the performance impact of Vortex’s object-oriented optimizations, we compare the following configurations:

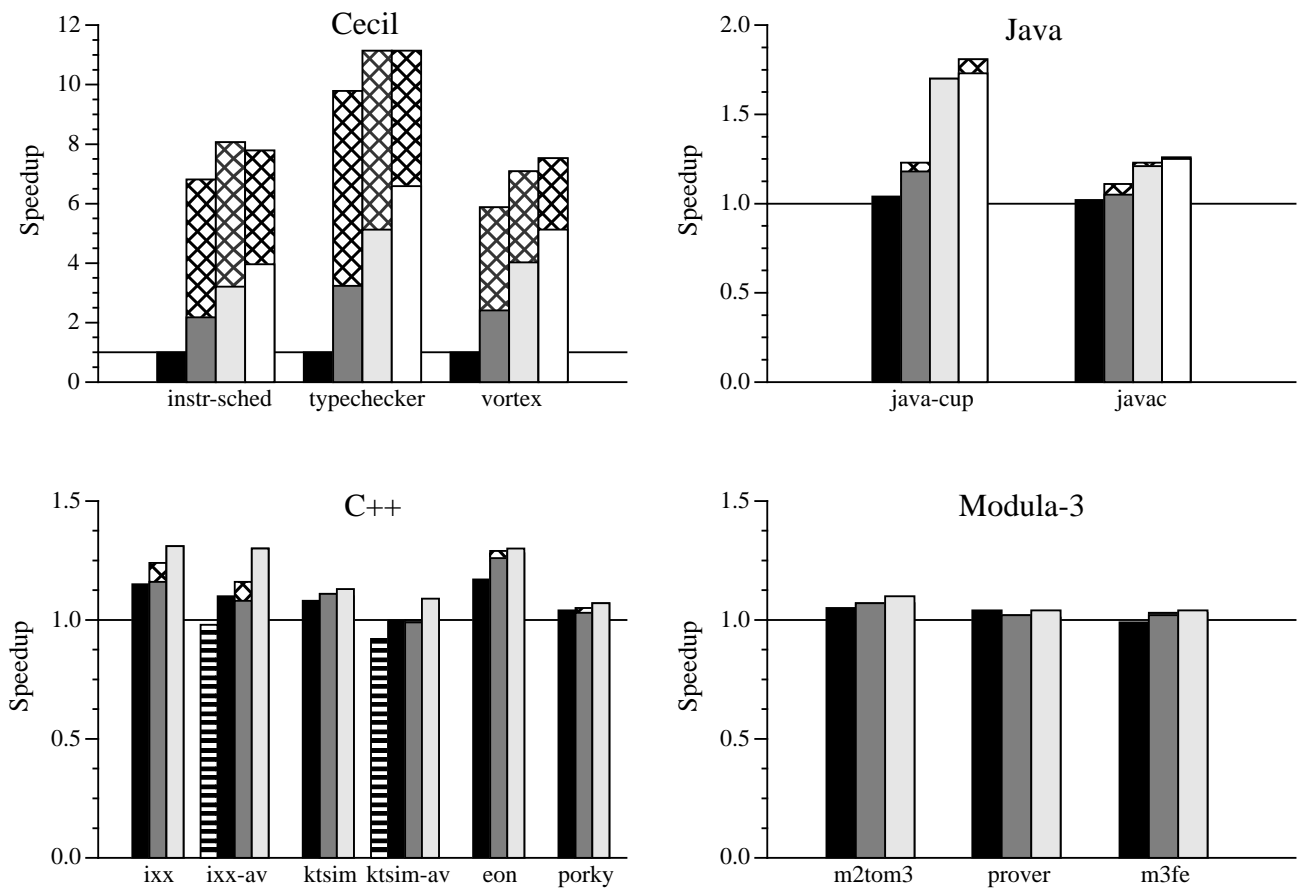
- **native**: Program is compiled by a native language compiler. For C++ programs we use **g++** version 2.6.3 with options **-O2 -finline-functions -msupersparc**; for Modula-3 programs we use the DEC SRC Modula-3 compiler (which uses a modified version of **gcc** version 2.6.3 as its back-end) with the **-O2** option. For Java the native configuration runs the **java** interpreter on the program’s precompiled **.class** files. There is no separate native configuration for Cecil programs, since Vortex is currently the only Cecil compiler.
- **base**: Program is compiled by Vortex with the collection of traditional optimizations described in Section 3.2 but without any inlining or optimization of dynamically-dispatched messages or class tests. (For C++ and Modula-3, the combined CSE pass is not fully functional,

so the **base** configuration in these languages uses only **gcc** optimizations with no Vortex- level optimizations.)

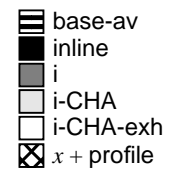
- **inline**: **base** is augmented with cross-module inlining.
- **intra**: **inline** is augmented with intraprocedural class analysis, hard-wired class prediction for common messages (for Cecil programs only), and splitting.
- **intra+CHA**: **intra** is augmented with class hierarchy analysis.
- **intra+CHA+exh**: **intra** is augmented with class hierarchy analysis and exhaustive class testing.
- **intra+profile**: **intra** is augmented with profile-guided receiver class prediction.
- **intra+CHA+profile**: **intra** is augmented with class hierarchy analysis and profile-guided receiver class prediction.
- **intra+CHA+exh+profile**: **intra** is augmented with class hierarchy analysis, exhaustive class testing, and profile-guided receiver class prediction.

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

All run-time measurements are CPU times (user time + system time) gathered on a lightly-loaded SPARCStation-20/61 with 128MB RAM, taking the median of 11 runs. Variations in CPU time from run to run of 10% are normal for this machine. Figure 2 presents selected application speedup data; the complete set of application execution times and the dynamic number of message sends during each program execution can be found in Appendix A.



**Figure 2:** These graphs show application speedup relative to the **base** configuration of the application. For the two all-virtual C++ programs, speedups are relative to the **base** configuration of the original application, and an additional bar, **base-av**, is shown. The impact of augmenting the **i**, **i-CHA**, and **i-CHA-exh** configurations with profile-guided class prediction is shown by the additional height of the cross-hatched portion of each bar. For some configurations, the impact of profile data was negligible, and thus is not visible above.



As expected, Vortex’s optimizations had the largest impact on the Cecil programs, improving their performance by an order of magnitude over the **base** configuration. Profile-guided class prediction was the single most important optimization for Cecil, but each technique resulted in a non-trivial improvement. Although less spectacular than the Cecil results, many of the C++ and Java programs obtained speedups on the order of 25-35%, with **java-cup** achieving an 80% speedup; **m2tom3** showed the largest speedup of the Modula-3 programs, gaining about 5% from cross-module inlining and an additional 4% from the object-oriented optimizations. In the non-Cecil programs, class hierarchy analysis was more effective than profile-guided class prediction.

The performance gains obtained by applying these optimizations do not come without cost; both compile time and compiled code space typically increase as a result. Compile time increases can be broken into two components: the direct cost of performing the analysis/optimization and the

indirect cost of increasing the amount of inlining, and thus increasing the amount of time spent in other compilation phases. For these programs, we found that the majority of the compile time increase was attributable to other compilation phases being faced with larger control flow graphs after inlining. In the worst case, observed for a subset of the non-Cecil programs, compilation time doubled between **base** and **inline**, and doubled again between **inline** and any of the other more optimized configurations. However, Vortex is a research compiler, designed for ease of extension rather than compilation efficiency, and so the magnitude of the compile time increases may not be indicative of what would be seen if these techniques were implemented in a production compiler. In contrast, code space costs were modest. For the C++ programs, compiled code space grew by 3% to 20% over the **base** configuration. For the other three languages, code space changes ranged from -10% to +4% over **base**.

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

In addition to improving the performance of existing applications, high-level optimizations like class hierarchy analysis and receiver class prediction can reduce the need for programmers to hand-optimize their programs. For example, the version of `ktsim` that was provided to us had been hand-optimized by manually removing all virtual function calls from the common execution paths. This version compiled without class hierarchy analysis performed slightly worse than the all-virtual version optimized automatically with class hierarchy analysis.

## 6 Related Work

Several other projects have worked on implementing and assessing advanced optimizations for hybrid languages: Calder & Grunwald, Aigner & Hölzle, Bacon & Sweeney, and Pande & Ryder have looked at optimizing C++, and Fernandez and Diwan, Moss, & McKinley have looked at optimizing Modula-3.

Calder and Grunwald consider several ways of optimizing dynamically-dispatched 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, suggesting that profile-guided receiver class prediction would pay off. However, they only simulated the effects of converting indirect calls to direct calls (ignoring other benefits such as inlining), and they did not test this hypothesis by implementing class prediction in a compiler.

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 system works by first combining a C++ program composed of multiple source files to produce a single, monolithic C++ file. This file is then fed into their optimizer, which works as a source-to-source transformation, producing another C++ file (that has been optimized with some combination of class hierarchy analysis and profile-guided receiver-class prediction). Finally, this single file is compiled with a native C++ compiler to produce the final program executable. One major difference with the Vortex approach is that they leave inlining to be done by the host C++ compiler, rather than performing it in conjunction with class hierarchy analysis and class prediction. This has the effect of preventing their preprocessor from analyzing the to-be-inlined callee in the context of its call site, and analyzing the rest of the caller using static class information derived from the callee. This lack of integration also biases their results on the effects of combining class hierarchy analysis with profile-guided receiver class prediction because the bodies of methods that are inlined via class hierarchy analysis do not benefit from class prediction, thus making class hierarchy analysis appear to be less effective than it actually would be in a fully integrated implementation. On the two benchmarks we have in common (`ixx` and `porky`), Vortex was able to eliminate a larger percentage of the virtual functions calls and achieved marginally better speedups. Finally, their system does not support selective recompilation, since their source-to-source transformation starts by combining the entire program into a single C++ source file.

Bacon and Sweeney examined the effectiveness of three purely static techniques for replacing virtual function calls with direct calls in C++, one used when there was only one method with a particular name (and argument type signature) in the program, class hierarchy analysis, and rapid type analysis (an extension of class hierarchy analysis that prunes unreachable classes and methods in parallel with computing the possible targets of call sites) [Bacon & Sweeney 96]. For two of their five non-trivial benchmarks, ranging in size from 5,000 to 17,500 lines, rapid type analysis statically-bound more calls than class hierarchy analysis. They did not look at the effectiveness of run-time class testing (driven by either class hierarchy analysis or execution profiles), nor did they report bottom-line impact on execution time due to their transformations. They do report that rapid type analysis shrinks the size of the generated programs substantially.

Pande and Ryder apply aggressive interprocedural context-sensitive pointer analysis to statically-bind virtual function call sites in C++ [Pande & Ryder 94]. Their results (in terms of the number of dynamically-dispatched call sites that are statically bound) appear good, but their current benchmark suite is made up of programs of less than 1000 lines, so it is unclear how the quality and cost of their analysis will scale to

larger, more realistic C++ programs. Also, they do not report the bottom-line run-time performance impact of their optimizations.

Fernandez developed an optimizing Modula-3 system that performs class hierarchy analysis, inlining, and procedure specialization at link-time [Fernandez 95]. Her system delays all code generation until link time: the compiler does a simple translation of each source file to an intermediate representation, and the linker combines these IR files and generates code for the entire program. Her optimizations were able to statically-bind between 2% and 79% of the indirect calls and to reduce the number of instructions executed by 3-11% in a suite of Modula-3 benchmark programs (compared to the output of an unoptimizing compiler). Because the linking step has become a bottleneck, her system performs only limited, basic-block level optimization; there is no comparison of her approach to that achieved using a more optimizing compiler. Our benchmark suites share the `prover` and `m3fe` programs. Fernandez reports that her version of class hierarchy analysis removes roughly 79% of the indirect calls in `prover` and 56% in `m3fe`, while we see only 20% and 0% drops, respectively, for our class hierarchy analysis. However, the DEC SRC Modula-3 front-end by default implements all cross-module procedure calls, even non-dispatched calls, as indirect jumps. To construct a more reasonable baseline system, we modified our Modula-3 front-end to implement regular Modula-3 procedure calls using direct jumps. Since Fernandez uses the unmodified Modula-3 front-end, her indirect jump reductions appear to include those for the trivially-optimizable cross-module procedure calls, overstating the effectiveness of her system at optimizing dynamically-dispatched method calls.

Diwan, Moss, and McKinley developed a whole-program optimizer for Modula-3 incorporating class hierarchy analysis (which they call type hierarchy analysis), intraprocedural and interprocedural class analysis, and a simple, monovariant heap analysis to compute class sets for instance variables [Diwan et al. 96]. They analyzed these algorithms on a range of Modula-3 programs, and their results indicate that class hierarchy analysis obtains nearly all of the benefit of the more powerful analyses (ignoring the possibility of sends to `NULL`, a distinguished object in Modula-3 that results in an error whenever a message is sent to it). They present performance results indicating less than a 2% performance increase, in part because their system does no optimizations based on the information computed by the analyses other than converting message sends into direct calls; in particular, they do not perform inlining. Our bottom-line speedup results for Modula-3 programs echo their limited gains, although our speedups appear slightly larger on the one benchmark (`m2tom3`) we

have in common. Diwan's system also does not support selective recompilation after programming changes.

Agesen and Hölzle compared the effectiveness of the Cartesian Product interprocedural class analysis algorithm [Agesen 95] with profile-guided receiver class prediction [Hölzle & Ungar 94] for Self programs [Agesen & Hölzle 95, Agesen & Hölzle 96]. In work related to our work in this paper, they extrapolated their results to a hypothetical hybrid language, Self++, by ignoring the influence of messages sent to objects that would be primitive datatypes in a hybrid language, concluding that the contributions made by both optimization techniques (in terms of percentage of dispatches eliminated) would be substantially reduced in hybrid languages. Our work can be viewed as real experimental data for a range of real languages to support their general prediction about hybrid languages.

## 7 Conclusions

Our studies using Vortex have provided some initial data on how well advanced optimizations for object-oriented languages impact the performance of sizeable programs written in a variety of languages, ranging from a very pure language (Cecil) to low-level hybrid languages (C++), with Java and Modula-3 providing intermediate points in the language design space. As shown previously, programs in dynamically-typed, purely object-oriented languages can speed up by an order of magnitude through the application of optimizations for message sends and closures. New results in this study show that programs written in statically-typed, hybrid languages can also achieve significant speed-ups, on the order of 25-35%, with little cost in compiled code space (although Vortex's research prototype nature leads to long compilation times). Our study benefits from using a common optimizing compiler back-end applying essentially the same suite of optimizations uniformly across all languages, providing a "level playing field" on which to compare the effectiveness of optimizations across programs and languages.

Vortex has been a good infrastructure for research on optimizations for object-oriented languages. By translating different languages into a common intermediate language that still retains high-level information such as the location of message sends, optimizations can be written once rather than separately for each language. The supporting dataflow analysis frameworks in Vortex make it much easier to add new optimizations. Vortex's dependency mechanisms to support selective recompilation help to make interprocedural or whole-program optimizations practical in a normal program development environment. In the future, we hope to add additional language front-ends (such as Smalltalk and ML) and additional optimizations (such as interprocedural class

analysis, alias analysis, and data representation and layout optimizations) to Vortex.

It may seem that Vortex is an ideal platform for comparing the performance of languages, for instance assessing how the performance of Cecil compares to C++. However, such cross-language comparisons are extremely difficult to perform well. First, realistically large programs need to be written in each language being compared; to date, we know of only small benchmarks of less than 1,000 lines that have been translated into several different object-oriented languages. Second, a decision must be made as to whether identical algorithms and language features should be used in the different languages (adopting a least-common-denominator style), or whether the programs should be written using the best or most common programming style for each language. The former strategy may produce more closely comparable results, but the latter strategy may better reflect the expected performance of the typical program in each language. We have not tried to perform cross-language comparisons to this extent. If such benchmarks were developed, however, Vortex might be an excellent test-bed for uniformly optimizing each of the language-specific versions.

## Acknowledgments

This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), an NSF Research Initiation Award (number CCR-9210990), a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM, Xerox PARC, Pure Software, and Edison Design Group. We would also like to especially thank the groups responsible for developing the basis for several of our language front-ends: Edison Design Group for donating their C++ Front End product, Digital Equipment Corporation's System Research Center for producing DEC SRC's Modula-3 implementation, and Sun Microsystems for producing the Java Development Kit. Urs Hölzle provided useful feedback on an earlier version of this paper. We are grateful to Gerald Aigner, David Detlefs, Amer Diwan, Mary Fernandez, Urs Hölzle, Dennis Lee, Ted Romer, and Peter Shirley for providing us with benchmark programs, and to Luca Cardelli for pointing us at Scott Hudson's java-cup program.

An initial release of the Vortex compiler and its Cecil front-end is currently available. We expect that the next release of the system, planned for late 1996, will include the C++, Modula-3, and Java front-ends. More information about the Vortex compiler is available via the World Wide Web at:

<http://www.cs.washington.edu/research/projects/cecil>

## References

- [Agesen & Hölzle 95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages. In *OOPSLA '95 Conference Proceedings*, pages 91–107, Austin, Tx, October 1995.
- [Agesen & Hölzle 96] Ole Agesen and Urs Hölzle. Dynamic vs. Static Optimization Techniques for Object-Oriented Languages. *Theory and Practice of Object Systems*, 1(3), 1996.
- [Agesen 95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.
- [Aho et al. 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [Aigner & Hölzle 96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *Proceedings ECOOP '96*, Linz, Austria, August 1996. Springer-Verlag.
- [AK et al. 89] Hassan Ait-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.
- [Bacon & Sweeney 96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA '96 Conference Proceedings*, San Jose, CA, October 1996.
- [Bieman & Zhao 95] James M. Bieman and Josephine Xia Zhao. Reuse Through Inheritance: A Quantitative Study of C++ Software. In *Proceedings of the Symposium on Software Reusability*. ACM SIGSOFT, August 1995. Software Engineering Notes.
- [Bobrow et al. 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, 28(Special Issue), September 1988.
- [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.
- [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 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehmman Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, June 1992. Springer-Verlag.
- [Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report TR-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.



- [Chambers et al. 95] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompile in the Presence of Complex Intermodule Dependencies. In *17th International Conference on Software Engineering*, Seattle, WA, April 1995.
- [Chambers et al. 96] Craig Chambers, Jeffrey Dean, and David Grove. Whole-Program Optimization of Object-Oriented Languages. Technical Report TR-96-06-02, Department of Computer Science and Engineering, University of Washington, June 1996.
- [Chien 93] Andrew A. Chien. *Concurrent Aggregates (CA): Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.
- [Click & Cooper 95] Cliff Click and Keith D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.
- [Dean et al. 95a] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. *SIGPLAN Notices*, pages 93–102, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.
- [Dean et al. 95b] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.
- [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.
- [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.
- [EDG] C++ Front End 2.28. Provided by Edison Design Group, Inc. <http://www.edg.com>.
- [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 Language and its Implementation*. Addison-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 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.
- [Hölzle & Ungar 94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *SIGPLAN Notices*, 29(6):326–336, June 1994. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
- [Hölzle 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.
- [JDK] Java Development Kit. Sun Microsystems Inc. <http://java.sun.com/>.
- [Johnson 88] Ralph Johnson. TS: AN Optimizing Compiler for Smalltalk. In *Proceedings OOPSLA '88*, pages 18–26, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.
- [Lea 90] Doug Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, San Francisco, CA, April 1990.
- [Nelson 91] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [Pande & Ryder 94] Hemant D. Pande and Barbara G. Ryder. Static Type Determination for C++. In *Proceedings of Sixth USENIX C++ Technical Conference*, 1994.
- [Plevyak & Chien 94] John Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings OOPSLA '94*, pages 324–340, Portland, OR, October 1994.
- [Schaffert et al. 85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment, Language Reference Manual. Technical Report DEC-TR-372, Digital Equipment Corporation, November 1985.
- [Shao & Appel 95] Zhong Shao and Andrew Appel. A type-based compiler for Standard ML. *SIGPLAN Notices*, pages 116–129, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.
- [SRC] DEC SRC Modula-3 Implementation. Digital Equipment Corporation Systems Research Center. <http://www-research.digital.com/SRC/modula-3/html/home.html>.
- [Stroustrup 87] Bjarne Stroustrup. Multiple Inheritance for C++. In *Proceedings of the European Unix Users Group Conference '87*, pages 189–207, Helsinki, Finland, May 1987.
- [Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addison-Wesley, Reading, MA, 1991.
- [Tarditi et al. 96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Bob Harper, and Peter Lee. TIL: A Type-Directed Compiler for ML. *SIGPLAN Notices*, pages 181–192, May 1996. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.
- [Tjiang & Hennessy 92] Steven W. K. Tjiang and John L. Hennessy. Sharlit – A Tool for Building Optimizers. *SIGPLAN Notices*, 27(7):82–93, July 1992. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.
- [Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87*, pages 227–242, December 1987.

## Appendix A Raw Data

**Table 3: Execution Time (seconds)**

Program	native	base	inline	intra	intra+ CHA	intra+ CHA+ exh	intra+ profile	intra+ CHA+ profile	intra+ CHA+ exh+ profile
instr sched		21.11	21.11	9.69	6.58	5.32	3.11	2.61	2.74
typechecker		333.82	338.82	103.44	64.79	50.55	33.84	30.08	29.78
vortex		3,617	3,617	1,500	903	700	615	515	486
java-cup	5.0	0.85	0.82	0.72	0.50	0.49	0.69	0.50	0.47
javac	62	10.21	10.02	9.70	8.43	8.17	9.17	8.32	8.08
m2tom3	23.50	23.42	22.29	21.87	21.36		21.91	21.92	
prover	28.90	32.75	31.54	32.25	31.55		32.02	32.39	
m3fe	21.90	22.78	22.91	22.28	21.90		22.16	22.13	
ixx	0.86	0.92	0.80	0.79	0.70		0.74	0.70	
ktsim	106.17	107.41	99.28	98.47	96.84		97.17	96.52	
eon	76.22	82.60	70.54	65.39	63.78		64.12	64.41	
porky	9.75	13.56	13.00	13.13	12.72		12.86	12.67	
ixx-av	0.88	0.94	0.83	0.85	0.71		0.79	0.71	
ktsim-av	113.60	116.67	107.73	108.24	100.60		107.42	98.98	

**Table 4: Dynamic Number of Message Sends (x1000)**

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