Expressive, Efficient Instance Variables

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

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

February 1996

Abstract

The decoupling of clients of abstractions from the implementations of those abstractions is a key benefit of object-oriented programming. However, many object-oriented languages provide instance variables in such a way that implementation-level representation decisions seep through interfaces, introducing coupling between clients and abstractions. As part of the Cecil language, we have developed a semantics for flexible instance variables based on the principle that a class should be decoupled from its clients, its subclasses, and its role extensions. We describe our design of instance variables and provide examples where their additional flexibility has been useful. However, implementing our semantics naively can impose significant performance penalties. To achieve reasonable performance, we have developed a number of optimizations that are effective at eliminating many of the costs of our flexible model. We discuss these optimizations and report on the results of experiments to assess the costs of our design under different optimization scenarios.

Keywords: Object-oriented programming, instance variables, language design, compiler optimization, Cecil, subject-oriented programming, roles

1 Introduction

The decoupling of clients of abstractions from the implementations of those abstractions is a key benefit of object-oriented programming. This decoupling allows implementations to be refined or even totally rewritten without affecting client code. Likewise, through the magic of subclassing and subtype polymorphism,[*] a single piece of client code can be written which can manipulate many different implementations of the same abstraction, potentially mixing these implementations at run-time. Dynamically-dispatched method calls are often the focus of this decoupling, but the treatment of instance variables by an object-oriented language also can affect the degree of decoupling and flexibility fostered by the language.

For the Cecil language [Chambers 93], we have designed a semantics for instance variables that supports a strong decoupling between a class and its clients, its subclasses, and its role extensions. Section 2 describes our guiding design principles and the resulting design of instance variables in Cecil, and section 3 provides examples and statistics on actual use of this instance variable design from our experience over several years in writing a large optimizing compiler in Cecil. However, the flexibility available with Cecil's semantics for instance variables can come at a steep cost in run-time performance. Accordingly, we have developed several key analyses and optimizations which greatly reduce the cost of our design for instance variables. In particular, when using instance variables in only a "normal" way, the performance of instance variable access is just as good as in C++ or Smalltalk; extra run-time cost is paid only when exploiting the extra flexibility of instance variables. Section 4 describes a straightforward implementation of Cecil's instance

[*] Subtype polymorphism allows an instance of a subtype to appear wherever an instance of a supertype is expected. In most (but not all) object-oriented languages, subtyping and subclassing are tied together, leading to a somewhat more restricted kind of polymorphism where an instance of a subclass is allowed wherever an instance of a superclass is expected.

variables and then describes several optimizations that improve the performance of this implementation. Section 5 presents some experimental results that indicate the cost of different aspects of Cecil's instance variables under different implementation strategies.

2 The Design of Flexible Instance Variables

2.1 Design Principles

The main principle guiding our design of instance variables is to support the decoupling of a class's implementation from its clients. Whether an operation is implemented via computation (a method) or via storage (an instance variable) should be considered an internal implementation decision, and as such hidden from clients so that this decision can be freely changed without affecting clients. By "client" we include external users of instances of a class, the subclasses of a class, and the class itself; all of these clients should be insulated from whether an operation is stored or computed, to the extent possible. This design principle implies that a method should be able to be replaced with an instance variable, and vice versa, and moreover that a method should be able to be overridden with an instance variable, and vice versa, all without affecting any of these clients. If a method overrides an instance variable, it should still be possible to use a "super" send to access the overridden instance variable from within the method, just as with an overridden method.

We also wish to support a kind of role-based or subject-oriented programming style [WB & Johnson 90, Reenskaug & Anderson 92, Harrison & Ossher 93, Ossher et al. 95, VanHilst & Notkin 96] where different roles or facets of a class's behavior can be factored out into separate modules which are composed together at link-time to form the full implementation of a class [Chambers & Leavens 94]. One module defines the class and its core functionality. Other modules can augment this core with additional more application-specific functionality, using the encapsulation and name scoping facilities of modules to avoid polluting the core class interface with a large number of specialized operations. The effect of roles is to support finer-grained forms of modularization and decoupling than is available in traditional object-oriented languages. For instance variables, we wish to allow instance variables to be factored out of a class into a role extension module, just as methods can be. Since constructors of a class may not be aware of all role extensions, instance variables in external roles need to be able to be initialized to reasonable values without requiring modifications to the class's constructors.

A few other principles and goals contributed to our final design for instance variables:

- We wish to allow free subclassing and role extensions. Any class can admit subclassing, and any class can admit role extension. (Some applications may wish to limit subclassing or role extension in selected circumstances to aid in reasoning about the program's behavior, but the default should be to allow subclassing and role extension everywhere). A subclass can be stored in an instance variable wherever a superclass is expected.
- We wish to support immutable instance variables as well as mutable ones. An immutable instance variable should be able to be given a value when an instance is created. (We distinguish between immutable and mutable local and global variables in a similar fashion.)
- We wish to support both dynamic and static typechecking of accesses to and initializations of instance variables.
- We wish to be able to ensure that all instance variables have been initialized properly before being accessed, either statically or through run-time checking.
- We wish to allow both per-object instance variables and shared "class variables," hiding the distinction from clients in the same manner as the difference between instance variables and methods is hidden.
- We wish to support multiple inheritance, with simple rules for conflicts.

In the remainder of this section, we describe our design of instance variables which meets these goals. The highlights are as follows:

- instance variables are accessed solely through automatically-generated reader and writer accessor methods, making instance variables and methods interchangeable;
- instance variable declarations can specify a default initial value, potentially computed in terms of the values of other instance variables of the object; and
- initial value expressions are evaluated for each inheriting object on demand when the instance variable is first accessed on that object.

Subsection 2.2 describes the part of the design dealing with accessing instance variables, and subsection 2.3 describes initializing instance variables. Each subsection concludes with a comparison to related work.

2.2 Accessor Methods

2.2.1 Field Declarations and Accessor Methods

The foundation of our approach to instance variables is to access them solely through automatically-generated accessor methods. A mutable instance variable declaration introduces a pair of reader and writer accessor methods, while an immutable instance variable declaration introduces only a reader accessor method. The only way to reference the contents of the instance variable is by invoking one of the instance variable's accessor methods. For example, the following pseudocode uses the `field` declaration to declare three instance variables; the `var` annotation indicates those instance variables that are mutable (by default, instance variables, local variables, and global variables are immutable):

```
class Rectangle {
   var field center;
   field length;
   field height;
   ...
}
```

These field declarations implicitly produce the following hidden memory locations and accessor methods (italics font indicates internal declarations and code that cannot be written by users):

```
class Rectangle {
   instance_var _center;
   method center() { return <self._center>; }
   method set_center(new_center) { <self._center> := new_center; }
   instance_var _length;
   method length() { return <self._length>; }
   instance_var _height;
   method height() { return <self._height>; }
   ...
}
```

2.2.2 Invoking Accessor Methods

Clients of the instance variables use regular message passing to invoke the accessor methods:

```
let r := new Rectangle(...);
print(r.length);
r.set_center(r.center + offset);
```

If message-sending syntax is designed to look the same as instance variable accesses, as with the `r.length` and `r.center` messages above, then there is no syntactic burden to reading the contents of an instance variable by sending a message. Similarly, the language can support a syntactic sugar for assignment-like messages. For example, in Cecil, the last statement above could be written as:

```
r.center := r.center + offset;
```

because an "assignment" of the form `e.msg := rhs` is syntactic sugar for the message `set_msg(e, rhs)`. As a result, code can be written to appear to manipulate instance variables directly, while actually sending messages which could invoke either accessors or regular methods at run-time.

The main advantage of accessing instance variables and methods uniformly through dynamically-dispatched messages is that clients (i.e., senders of messages) are insulated from decisions as to whether a given operation will be implemented by an instance variable or by a method. The implementation of `Rectangle` above could be changed to store an `upper_left` instance variable instead of `center`, and no clients would need to change as long as `center` and `set_center` methods are left in the field's place:

```
class AlternateRectangle {
   var field upper_left;
   -- reimplement center "field" as a pair of methods:
   method center() {
      return self.upper_left + new_point(self.length/2, self.height/2); }
   method set_center(new_center) {
      self.upper_left := ... new_center ... }
   field length;
   field height;
   method area() { return self.length * self.height; }
   ...
}
```

By requiring all clients, even the class itself and its subclasses, to access instance variables only through the accessor methods, we help to reduce the cost of changing such implementation decisions.

2.2.3 Accessor Methods and Inheritance

A second advantage of accessing instance variables solely through accessor methods is that many of the standard issues in object-oriented languages can be made to apply uniformly to both methods and instance variables, and solutions invented previously for methods can be adopted for instance variables. For example, if the language supports encapsulation of methods within a class, then encapsulating accessor methods in the same way enables instance variables to be encapsulated. The reader and writer accessor methods can be given different visibilities, such as making the reader method public while protecting the writer method, a flexibility usually omitted for traditional instance variables. Static typechecking of accesses to instance variables can be derived directly from static typechecking of regular method invocations. Finally, rules developed in the context of methods for inheritance, overriding, and handling multiple inheritance name clashes can be directly adapted to provide elegant, consistent, and flexible solutions for instance variables. For example, it becomes easy to override a method with a field and vice versa: it is the accessor methods that are overridden (using regular method overriding rules), not the internal memory location:

```
class DisplayedRectangle subclasses AlternateRectangle {
   var field lower_right;
   -- override length and height fields with methods:
   method length() { return self.lower_right.x - self.upper_left.x; }
   method height() { return self.lower_right.y - self.upper_left.y; }
   -- override set_center to also update lower_right:
   method set_center(new_center) {
      self.lower_right := ...;
      resend; } -- invoke set_center of superclass
   -- override set_upper_left to additionally redisplay the moved rectangle:
   method set_upper_left(new_upper_left) {
      resend; -- invoke set_upper_left accessor
      self.redisplay; }
```

```
    -- override area method with field:
    field area;
}
```

Even if a field's accessor methods are overridden, however, the field's memory location may still be accessible. For example, an overriding method may perform a `resend` (akin to a `super` send in Smalltalk) that invokes the overridden accessor method, as with the `set_upper_left` method above. Of course, if none of the overriding methods performs a resend, then the accessor methods are unreachable, and an optimizing compiler could eliminate the memory allocated to the unreachable field. Similarly, language extensions like Eiffel's `undefine` clause [Meyer 92] could be used to explicitly remove a field's accessor methods from the set of methods inherited by a class, thereby rendering the field's memory unreachable and reclaimable.

In the presence of multiple inheritance, a common semantic issue is how to handle name clashes, both for methods and for instance variables. Our approach to instance variables reduces the problem to that of resolving clashes among methods. Memory locations for instance variables do not clash, but their accessor methods can. In Cecil, we make it a static programming error for two different methods with the same name to be inherited by a single class, and Cecil provides an explicit mechanism for programmers to insert a resolving method in the subclass that resends the message as appropriate to any of the inherited versions of the method; other languages may adopt different rules for managing name clashes, such as the linearization policy of CLOS [Paepcke 93]. When instance variables are accessed solely through messages, the policies for method name clashes directly give a semantics for field name clashes. In Cecil, for example, if two fields with the same name (or a field and a method with the same name) are inherited by a single class, the programmer must insert explicit methods to resolve the ambiguity. The memory for the two fields is still accessible in the class; the overriding method can choose to invoke one or both of the fields' accessor methods. For diamond-shaped inheritance hierarchies, with a single class being inherited along multiple paths, Cecil adopts an inheritance semantics where only a single version of methods inherited from the top of the diamond are seen in the bottom of the diamond; for field accessor methods, this choice implies that only one copy of the memory for the instance variable will appear in the class at the bottom of the diamond. Other languages, such as C++ and Eiffel, support inheritance semantics that allows replication of state along multiple paths of the inheritance graph. In our model of instance variables, this would be accomplished by considered that methods are replicated along multiple paths, continuing the parallel semantics between methods and fields.

2.2.4 Role Extensions and Accessor Methods

For role-based programming, methods can be added to pre-existing classes from the outside, to augment the interface of standard classes for a particular application's needs and to reduce the pressure to add all possible operations to the original class. In Cecil, these extensions can be individually encapsulated in modules. By treating fields and methods uniformly, we enable instance variables to be added to pre-existing classes, too, with little additional semantic complexity.

2.2.5 Shared Fields

Several languages support a notion of class variables, which are shared among all the instances of a class. Cecil supports a similar notion, a shared field. Shared fields are treated very much like regular object-specific fields; in particular, accessor methods are generated and invoked for shared fields just as for regular fields. The only difference between shared and regular fields is that the memory location allocated for a shared field is allocated statically rather than in each object. All invocations of a shared field accessor method reference the same memory location, independent of the object on which the accessor method is invoked. Again, the decision as to whether an operation is implemented as a shared field, a regular field, or a method is hidden from clients.

2.2.6 Related Work

Our approach to introducing accessor methods to unify instance variables with methods was inspired by the similar treatment of instance variables included in Trellis [Schaffert et al. 85, Schaffert et al. 86] and Self [Ungar & Smith 87]. Trellis's approach is essentially the same as our design, with regard to accessor methods, support for read-only instance variables, overriding methods with fields and vice versa, and multiple inheritance. Self is similar, but because of its prototype-based semantics it does not inherit an instance variable declaration but rather inherits the parent object's instance variable state. The Self programming environment has a mechanism to support copying of instance variable declarations from superclasses to subclasses [Ungar 95], but this mechanism does not work well when overriding an instance variable with a method that does a resend to access the overridden instance variable. Dylan [Dyl92] and Sather [Omohundro 94] also access instance variables via messages. CLOS [Bobrow et al. 88, Gabriel et al. 91] provides linguistic support for automatically generating reader and/or writer accessor methods for any of a class's instance variables, but it is always possible to access an instance variable directly, using `slot-value` or `with-slots`. Eiffel [Meyer 92] accesses both instance variables and methods uniformly with messages, but Eiffel does not allow an instance variable to be overridden with a method.

In object-oriented languages where instance variables and methods are distinct, programmers can always adopt conventions to write their own accessor methods. However, this convention can be tedious to follow, and widespread application of this idiom can severely hurt run-time performance, given standard implementation techniques for message passing.

2.3 Field Initialization

Instance variables are not only read and written, but also initialized. We treat initialization of an instance variable as a different activity than writing an instance variable, in part because we wish to support immutable instance variables which are initialized but not updated. In addition, our desire to support role-based programming, where instance variables can be added to pre-existing classes externally, requires special treatment of initialization separate from regular class constructors.

2.3.1 Default Initializers and Object Creation Primitives

In our design, an initial value for an instance variable can be specified either as part of the special object-creation primitive or as part of the field declaration:

```
class Rectangle {
   var field center := new_point(0, 0);
   field length := 10;
   field height := self.length; -- default to a square
   ...
}
let r := new Rectangle(length := 25);
print(r.area); -- prints 25^2 = 625
```

In this example, default initial values are given for each of the fields. The creation of the `r` instance provides a particular value for the `length` field but uses the defaults for the other fields of the rectangle. Even though `length` is immutable, it can be given an initial value whenever an instance is created; in fact, immutable instance variables must be given values either upon object creation or through a default value as part of the field declaration. Shared fields cannot be initialized as part of object creation; an immutable shared field must be given its value as part of its field declaration. The default initializer of a non-shared field is evaluated separately for each object created, if needed (as described in section 2.3.5 below).

2.3.2 Self-Reference in Initializers

The default initial value for a field is given by an arbitrary expression. This expression is allowed to reference the object containing the field (through the name `self` in our pseudocode). This mechanism

allows a field's default value to be expressed in terms of the values of other fields of the object. In the example above, the `height` of a particular rectangle by default is the same as the `length` of that same rectangle.

2.3.3 Initializers vs. Constructors

To create an object, Cecil includes an atomic object creation primitive. This primitive includes an optional list of initial bindings for the fields of the created object. Coupled with the ability to provide default initial values for fields, particularly defaults referencing `self`, the need for specialized non-trivial constructor methods is greatly reduced. Additionally, this approach is crucial for supporting immutable fields that are given their values at the point of object creation. Even so, our normal programming practice is to define separate creator methods that encapsulate the representation of objects (which would otherwise be exposed by the object creation primitive):

```
method new_rectangle(c, l, h) {
   return new Rectangle(center := c, length := l, height := h) }
method new_square(c, l) {
   return new Rectangle(center := c, length := l) }
```

Unlike constructors in most object-oriented languages, these creator methods are regular user-defined methods that can do arbitrary computation before and after instantiating a class. The creator method is under no obligation to return a new, direct instance of the named class; any subclass could be returned, or a pre-existing instance could be retrieved from a cache and returned, or any other object could be returned that supports the interface expected by clients of the creator method.

One weakness of our use of an object creation primitive as opposed to special constructor methods is that it is difficult to inherit initialization code. For example, if a subclass of `Rectangle` is defined that adds some instance variables, the code to create an instance of this new class often must repeat much of the code to initialize the superclass:

```
class ColoredRectangle {
   field color := Black;
   ...
}
method new_colored_rectangle(c, l, h, clr) {
   return new ColoredRectangle(center := c, length := l, height := h,
                                                          color := clr) }
```

If the code to initialize an instance of `Rectangle` were more complex, then this code would likely need to be repeated in each subclass of `Rectangle`. We are investigating alternatives that would allow better reuse and sharing of object initialization code.

2.3.4 Role Extensions and Initialization

Field declarations with default initial values are important to support role-based programming in our approach. Fields need to be added to separately-implemented classes without modifying the existing creator methods of the class, but the fields also should be given reasonable initial values, and immutable fields should be able to be added as separate role extensions. In our approach, externally-defined fields rely on default initial value expressions to get their initial values. It becomes particularly important in this context to support `self` references in initialization expressions, so that an externally-defined field can compute its initial value from the values of other fields of the created object (actually, in terms of any operations of the object visible in the scope where the field is declared).

One weakness of our current approach is that it becomes difficult to add state externally that is transferred appropriately when an object is copied. With our current mechanism, unless the copy method is somehow modified, the copy will get a fresh new value for its external fields, rather than some function of the field's value in the copied object. We are investigating adding a "copy accessor method" for each field that

computes the value of a field in the copy as a function of the value of the field(s) in the copied object. The default copy accessor method would be the identity function, leading to a shallow-copy semantics. In essence, we would be decomposing the copy operation into separate default pieces of functionality associated with each field, analogously to decomposing the initialization behavior for an object into separate default initialization expressions for each field, which better supports the decentralized programming model of role-based programming.

2.3.5 Lazy Evaluation of Initializers

With the initialization code for an object scattered across its field declarations and the creation primitive, the order of evaluation of the initialization code becomes an issue. In particular, field initializers containing `self` references should only be evaluated once the fields that they access have been initialized. Textual order makes little sense, since it would likely make `self`-referential initialization awkward, and since there may be no notion of the relative order of externally-defined fields in roles. Consequently, we chose a demand-driven evaluation order for field initializers. When an object is created, its explicit initializations provided as part of the object creation primitive are executed; all other fields are given the distinguished initial value UNINITIALIZED. If a reader accessor method is invoked for a field whose value is UNINITIALIZED, then the field's default initialization expression is evaluated. (This evaluation may in turn trigger nested evaluations of other fields' default initializers. Recursively invoking the reader accessor method on the same object is a run-time error, since it usually would lead to infinite recursion.) When the field's initializer returns a value, the field's contents is updated to record this initial value, and the value is returned as the result of the accessor method. Future invocations of the reader accessor method simply return the contents of the field. Essentially, the order of evaluation of field initializers is defined by the underlying (acyclic) data dependencies among the field values.

In addition to resolving the issue of evaluation order, demand-driven field initialization offers some other important advantages. If a field is never referenced, then its initializer is never executed. This can deliver some of the advantages of lazy functional languages with conceptually infinite data structures and streams. Also, checking for accesses to uninitialized fields becomes a special case of lazy evaluation of initializers: if a reader accessor method is invoked on a field whose value is UNINITIALIZED, and there is no default initialization expression, then a run-time error is reported. Often, the initializer expression for some field cannot run successfully until other computation and updates to the object have taken place; lazy evaluation enables the client to avoid executing the initializer by not referencing the field until its other state has been established.

A common idiom in our programming is to use fields as memoizing caches for expensive but idempotent methods. For example, if the `area` computation for some shape were expensive, a field could be defined as the external interface to an underlying `compute_area` method, which caches the computed area to quickly service future `area` requests:

```
class Ellipse {
   ...
   field area := self.compute_area;
   method compute_area() { ... }
}
```

Through lazy evaluation, the `area` field will be initialized at most once. If no client needs the `area` of a particular ellipse, then the expensive `compute_area` method will never be run. But if `area` is frequently accessed, the cost of a single `compute_area` invocation will be amortized over all `area` requests.

Aggressive use of this idiom would suggest that it would be useful to be able to "flush" the field's cache, resetting the state of the field to UNINITIALIZED, so that the next reference to the field will recompute its value. For example, if the shape of the ellipse is mutable, then `compute_area` is no longer idempotent, and it would be necessary to be able to forget any previously cached values for `area` and recompute the

new area the next time it is needed. For the moment, we have been hesitant to support this flushing mechanism, for fear that the semantics of fields would become too overloaded and the notion of an immutable field would be weakened.

2.3.6 Related Work

Cecil's field initialization mechanism follows most closely the CLOS approach. CLOS allows the initial values of instance variables to be provided as part of object creation (using `:initarg`) or through a default expression evaluated with the object is created (`:initform`). Self and Java both allow default initializer expressions to be given with instance variable declarations, but in Self's case the initializing expression is evaluated at the time of declaration. Dylan supports all three variations: default evaluated at declaration time, default evaluated at object allocation time, and value provided as part of the object creation primitive. A few languages, such as Modula-3, Sather, and Eiffel, allow instance variables to be given a default value at the time of declaration, but only if the value is a compile-time constant. However, none of these languages supports lazy evaluation of default initializers: either the initializer is executed at compile-time, at declaration-time, or at some point during object allocation. Similarly, none of these languages supports `self` reference in the initializer.

Cecil's model exploits lazy evaluation of initializers to cheaply detect accesses to uninitialized instance variables. Other languages either specify a default value or set of possible default values for an otherwise uninitialized field or leave the semantics of such accesses unspecified.

For object creation, most languages seem to fall chiefly into two camps. One camp, including Cecil, Modula-3, and CLOS, provides a special primitive atomic operation to allocate a new instance, optionally providing initial values for some of the instance variables of the created object (Modula-3 even allows object-specific method bindings to be provided as part of object creation). The other camp, including C++ Smalltalk, Trellis, and Eiffel, provides a specialized constructor operation that allocates space for an object and then runs user-specified code that performs assignments to the object to initialize its instance variables. Self is more in the second camp, although the basic allocation primitive performs a shallow-copy of some existing object, potentially avoiding some initializations due to copying a pre-initialized object. The advantage of the atomic allocation primitive is that it becomes easier to model immutable instance variables, since assignments are not used for initialization. However, it appears to be more difficult to inherit initialization code in this approach.

3 Use of Flexible Instance Variables in Practice

This section discusses our experience in writing software using this model of instance variables, and presents some real-world examples where we have found the flexibility afforded by this model useful. For the past two years, we have been developing Vortex, an optimizing compiler for object-oriented languages. Vortex is written entirely in Cecil and currently contains about 65,000 lines of code, including 1176 classes[*], 7726 methods, and 1262 instance variables. The code makes heavy use of many of the features of Cecil's instance variables. In writing Vortex, we have found several aspects of Cecil's instance variables especially useful:

- *Uniformity of fields and instance variables*. We have come to appreciate the uniform appearance of fields and methods in our interfaces, since it allows us to redesign implementations without affecting clients (where client refers to traditional external clients, subclasses, and the class itself). As a concrete example of this, early in the development of Vortex, we had a system that would perform whole program optimizations by examining aspects of the source code for the entire program [Dean et al. 95]. Optimization passes query various program representations to determine information, often maintained

[*] Cecil is actually a prototype-based language [Chambers 93]. For clarity, however, we use the more common terminology from class-based languages.

in publicly-readable instance variables, and then use the results to perform optimizations. However, the system initially did not keep track of dependencies introduced across source modules, and therefore did not support incremental recompilation: clients would freely access these instance variables and make optimization decisions that depended on their contents not changing. In order to support incremental recompilation, we renamed such instance variables and replaced them with methods that returned the contents of the renamed instance variable, but also recorded that the module being compiled now depended on the information. If the information changed, these dependencies indicate what compiled modules went out of date as a result of the change [Chambers et al. 95]. This was easy to implement because of the uniformity of instance variables and methods, making the switch from field to method completely transparent to clients.

We also use the ability to override fields with methods and vice versa: of our 1262 instance variables, 27 of them override methods and another 19 of them are overridden by methods. Usually the reasons that we override a method with a field are performance-related: a method that recomputes some information is inherited from a superclass, but in the subclass the time/space tradeoffs are different, and it is decided to compute the value once and to cache the answer in an instance variable. The opposite case of overriding an instance variable with a method occurs, we have found, when it is desirable to execute code as a result of a state change. For example, our compiler automatically generates code to initialize certain runtime system data structures, and the nature of this code changes when certain compiler options change. To ensure that we correctly regenerate this code, we override the writer accessor messages of fields that hold these compiler options to also invalidate the initialization code and then do a `resend` operation to invoke the writer accessor for the field.

- *Support for role-based programming*. The ability to define interfaces and implementations that have a small set of core functionality, and then to extend this functionality in other modules, has been a helpful means of organizing some of our code. As an example, our compiler defines a call graph abstraction, with call graph nodes representing methods in the program, and edges in the graph representing caller-callee relationships. Various modules extend this core functionality in different ways. For example, one module extends call graph nodes and edges by adding instance variables to store profile data for the program. Other modules extend call graph nodes various kinds of interprocedural summary information. By factoring our call graph abstraction into its core functionality along with various role-based extensions, the code becomes easier to understand. In examining our code, we find that roughly 30% of our methods are defined in modules external to where their specializing classes are defined[*] (2421 of 7726 methods), but less than 3% of our instance variables are defined externally (31 of 1262 instance variables). We suspect that this relative rarity of externally defined instance variables is due to two factors. First, as programmers, we often want to be able to know what state maintained by a class. We currently do not have programming environment support to provide a view of a class with roles collapsed, and so we tend to define instance variables in the same module as the class to which they are attached. Second, for classes that support `copy` methods, Cecil's current object creation expression semantics forces us to modify the body of the `copy` method whenever we want to provide a value for a field that depends on the value of the field in the original object. This gives a disincentive to define fields externally, since one has to go to the original source module to modify the `copy` method. With better linguistic and programming environment support, we believe we would be much more likely to define instance variables in external role extensions.
- *Lazy initialization of fields*. Lazily-initialized fields have proven very useful. First, because the initialization expressions can refer to other fields of the object, programmers can often avoid writing complex methods (constructors) to create instances of a class. Of our 1262 instance variables, 641 of them have initialization expressions; 55 of these initialization expression refer to `self` to compute their

[*] Cecil incorporates multi-methods, so we define a method as being "externally defined" when its definition occurs outside of the definition modules for any of the classes on which the method is specialized.

initializing value. Second, as discussed in section 2.3.5, lazily-initialized fields provide an easy means of defining a memoizing cache of some expensive computation. For example, our compiler decides whether or not a method is a promising candidate for inlining using a cost function that examines the abstract syntax tree (AST) for the method's body to estimate the code space and compile time cost of inline expansion. Computing this cost function involves examining each node in the method's AST. Initially, we had written this cost function as a method, but for routines that are frequent candidates for inlining, recomputing this cost function turned out to be expensive. By simply changing the method to a field whose initialization expression evaluated the cost function, we were able to memoize the result. This lazy evaluation and memoization could be simulated manually by the programmer in a language that lacked lazily-initialized fields, but the direct linguistic support makes this kind of transformation easier.

4 The Implementation of Flexible Instance Variables

In section 4.1 we develop in several stages one possible scheme for implementing instance variables that supports all of the expressive features described above. Unfortunately, supporting this flexibility could entail large runtime overheads. Therefore in sections 4.2 through 4.5 we examine a variety of optimizations that attack the various potential sources of runtime inefficiency.

4.1 Implementing Expressive Instance Variables

In general, the implementation translates a field declaration into a declaration of a memory location and an associated set of accessor methods that allow clients to read and write the memory location. All accesses to the instance variable occur via dynamically-dispatched message sends to these accessor methods. An initial implementation attempt simply declares the memory location and generates reader and writer accessor methods that load/store a value at a fixed offset into the receiver object. Using this approach, we would generate accessor methods similar to the templates shown below:

```
class A {
   method field1() { return ACCESS(self, OFFSET); }
   method set_field1(val) { ACCESS(self, OFFSET) := val; }
}
```

where `OFFSET` is the statically-computed constant offset of the accessor's associated memory location, and `ACCESS(x, y)` represents accessing the y^{th} field of object `x`. Depending on whether `ACCESS` appears on the left or right hand side of an assignment, it can denote either writing or reading an instance variable. This simple translation relies on the ability to compute a value of `OFFSET` that will be valid in all classes that inherit these accessors.

Unfortunately, in the presence of multiple inheritance, it is not always feasible to assign an instance variable the same offset in all inheriting objects. One straightforward extension that allows instance variable offsets to change in inheriting objects is to generate an additional pair of accessor methods at those classes where the offsets change. However, this extension cannot easily be reconciled with allowing the user to freely override instance variables with methods. For example, suppose the user overrides a field with a method in an intermediate class between the ancestor class that declares the field and a descendant class where its offset changes. In this second case, just defining additional accessor methods attached to the child class would alter the semantics of the program, since the new accessor method would hide the definition of the instance-variable-overriding method. To avoid these potential complications, we introduce an additional method that returns the offset of the instance variable. The hard-wired constant offset in the reader and writer accessors is replaced by a dynamically-dispatched message send to select the appropriate offset method. Since these offset methods are user-invisible, we can freely define them at those points in the class hierarchy where offsets change without worrying about them interfering with user-defined methods. With offset methods, our template accessors would look like:

```
class A {
   method field1() {
      offset := self._offset_field1;
      return ACCESS(self, offset);
   }
   method set_field1(val) {
      offset := self._offset_field1;
      ACCESS(self, offset) := val;
   }
   method _offset_field1() { return OFFSET_field1_in_A; }
}
class B inherits A {
   method field1() { ...resend...}
}
class D inherits B, C {
   method _offset_field1() { return OFFSET_field1_in_D; } ⇐ offset changed due to MI
}
```

Generation of offset methods is done during object layout. If fields are only allowed to appear in class declarations, then object layout can be done at compile time. However, if fields might be declared outside of class declarations, then we delay object layout until link-time when all fields become available to the implementation.

We now extend this basic field implementation to include support for lazily-initialized fields. First, we need to set aside a unique value to indicate that the field has not yet been initialized; in the discussion below, this value is denoted as UNINITIALIZED. For language implementations that use tagged representations, defining UNINITIALIZED is straightforward: one simply picks an invalid pointer value and tags it as a pointer. In the absence of tagging, it is still possible to track uninitialized fields. For example, each object's header could include a bit-vector that indicated which fields have been initialized. Once a scheme for denoting UNINITIALIZED values has been selected, the runtime system's object allocation routines can be modified to initialize all fields of allocated objects to this state.

Reader accessors are modified to check the loaded value against UNINITIALIZED. If the instance variable has not yet been initialized then the value of the field's default initializer is computed and stored. Note that this computation may reference the current values of the object's other instance variables by sending the appropriate accessor messages to `self`. An example reader accessor for a lazily-initialized field is shown below; the writer and offset methods are unchanged.

```
   method field2() {
      offset := self._offset_field2;
      value := ACCESS(self,offset);
      if (value = UNINITIALIZED) {
         value := <init_expr>;      ⇐ Computation of default initialization expression
         ACCESS(self,offset) := value;
      }
      return value;
   }
```

4.2 Optimizing Dynamic Dispatches

The previous section described a straightforward implementation of expressive instance variables. However, without optimization this implementation introduces large amounts of overhead, since each

access to an instance variable will result in at least two dynamically-dispatched messages. To reduce this overhead, we employ several well-known techniques for eliminating dynamic dispatches. The next subsections discuss several of these techniques, focusing on their effectiveness at eliminating the types of dynamic dispatches introduced by the implementation described above.

4.2.1 Intraprocedural Optimization

The compiler can replace a dynamic dispatch with a static call whenever it can determine that for all possible receiver classes, exactly one method will be invoked as a result of the dynamic dispatch. One intraprocedural technique for computing the set of possible classes is iterative static class analysis, which uses a standard iterative dataflow approach to determine the set of classes that correspond to program expressions [Chambers & Ungar 90]. The analysis maintains a mapping of variables to sets of classes, similar in spirit to the mapping of variables to constants that is used to perform constant propagation. When a dynamically-dispatched message send is encountered, the compiler utilizes this mapping to perform compile-time method lookup, thus computing the set of possibly-invoked methods. If only a single method could be invoked, then the dynamic dispatch can be replaced with a static call, which may subsequently be inlined. It is fairly common to send a number of messages to an object immediately after it is created, and these are easily optimized with iterative class analysis. However, due to its limited scope, intraprocedural optimization is only able to remove a relatively small fraction of the dynamic dispatches introduced by instance variable accesses.

Another local technique that can be used to statically bind message sends is profile-guided receiver class prediction [Hölzle & Ungar 94, Grove et al. 95]. It utilizes profile-derived frequency distributions to identify message sends that are dominated by a small number of receiver classes. It then optimizes for these common cases by inserting explicit class tests and inlined code for common classes while leaving a dynamic dispatch to handle those classes that are not explictly tested.

4.2.2 Whole-Program Optimization

When the compiler is able to expand the scope of its analysis to consider the entire program as a unit, it can more effectively eliminate the dynamic dispatches and other overheads associated with expressive instance variables. Whole-program optimizations can be broken into two categories: those such as class hierarchy analysis [Dean et al. 95] which only require access to all the declarations in the program, and other more intrusive optimizations such as cross-module inlining or full interprocedural class analysis that additionally require access to all method bodies.

Under class hierarchy analysis, when the compiler is able to examine all declarations in the program, there is no practical difference between fields defined in the class declaration and those defined external to the class declaration as part of a role extension. Therefore all overhead associated with role extensions can be eliminated, since both instance variable layout and size can be computed at compile time for all objects. Furthermore, the compiler can examine the class hierarchy and determine both where field offsets change, and where fields are overridden by methods. In a statically-typed language, applying class hierarchy analysis removes all instance-variable-related dynamic dispatches except where the programmer has taken advantage of the language's full flexibility, either by using multiple inheritance or by overriding fields with methods or vice versa. Class hierarchy analysis is also very effective in dynamically-typed languages, but without the additional information provided by static type declarations, it cannot always eliminate the dynamic dispatch invoking a reader or writer accessor.

Because of its global scope, whole-program optimization can introduce non-local dependencies in the compiled code, thus preventing 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 edited [Chambers et al. 95].

4.3 Optimizations for Lazily-Initialized Fields

As discussed in Section 2.3.1, in our design, an initial value for an instance variable can be specified either as part of the special object-creation primitive or as part of the field declaration. Values specified as part of the object creation primitive are evaluated eagerly and are stored directly in the fields when the object is created. The remaining fields of the object might be given values through lazy evaluation of the field's initialization expression, and therefore, to distinguish uninitialized fields from those fields that have already been initialized, the values of these fields must be set to a distinguished UNINITIALIZED value when the object is created. Supporting lazily-initialized fields thus imposes two costs. The first is this initialization of fields to the UNINITIALIZED value during object creation. The second is the check for the UNINITIALIZED value that is performed are part of the reader accessor method.

These costs are illustrated in the following code fragment:

```
      class Point {
         field x := 0;
         field y := 0;
      }
      p := new Point(x := 6); ⇐ Stores 6 to x and UNINITIALIZED to y
   ...
      return p.y;          ⇐ Loads p.y, compares with UNINITIALIZED, and if equal, initializes p.y to 0
```

Often, the compiler can optimize away the overhead of lazily-initialized fields through two simple techniques: *eager evaluation of simple initializers* and *detection of always-initialized fields*. Eager evaluation of simple initializers relies on the observation that if a field initialization expression is idempotent, side-effect free, and terminating, then it can be evaluated eagerly when the object is created, rather than storing UNINITIALIZED into the field upon object creation and initializing the field on its first reference. In general, identifying whether an initialization expression meets these conditions is undecidable, but field initialization expressions that are constants, or that name global objects form an easily-identifiable special case. Our compiler recognizes this case and copies down such simple initializers into object creation primitives. In the above example, the initializer for field `y` is simple, so eagerly evaluating it is safe, and this optimization would transform the object creation site into the following:

```
      p := new Point(x := 6, y := 0); ⇐ Stores 6 to x and 0 to y
```

This technique avoids the overhead of storing UNINITIALIZED into fields with these simple initializers. It can be applied to any field declarations that are visible to the compiler at the point that an object is created, and therefore does not rely on any form of whole-program knowledge. The more field declarations that are visible, the more often this technique will be applicable, however, so a whole-program view is useful, especially with external role-based extensions. This technique has some similarities to the strictness analysis employed by compilers for lazy functional languages [Hudak & Young 86], in that they both seek to determine when it is safe to eagerly evaluate expressions. The major difference is that strictness analysis examines the uses of an expression to determine when its value will definitely be needed later in the program, while our optimization examines the definition to determine when an initialization expression is safe (and profitable[*]) to evaluate eagerly.

[*] Even if an initialization expression can safely be evaluated eagerly, it is not always desirable to do so. For example, some initialization expressions allocate large data structures, and eagerly evaluating them can unnecessarily increase a program's memory usage if the field would not have been referenced during the object's lifetime.

The second optimization, *detection of always-initialized fields*, eliminates the check for UNINITIALIZED from the reader accessor method for fields that can be shown to always be initialized. There are two forms of this optimization. If a field declaration is visible to all allocation sites and has a simple initializer, then when teamed with eager evaluation of simple initializers, it is known that the field will always be initialized and the check for the UNINITIALIZED value can be eliminated from the reader accessor for that field. Field declarations are visible to all allocation sites either in the absence of roles, or in the presence of class hierarchy analysis, where all declarations in the program are visible to the compiler. A stronger form of this optimization can be applied when the compiler has access to the source of the entire program. In this case, all the object creation primitives in the program can be examined to identify those fields that are always given an initial value, increasing the number of fields for which the UNINITIALIZED checks can be elided.

4.4 Object Layout in the Presence of Multiple Inheritance

In the presence of multiple inheritance, it is not feasible to assign a single offset for a field that will hold for all subclasses. Because our implementation relies on dynamically-dispatched offset messages to determine the offset of a field within a particular class, there are no constraints that the layout of a subclass be anything like the layout of its parent class(es). Under class hierarchy analysis, however, the dynamic dispatch to compute the offset can be statically bound and the offset method inlined when the offset of a field is the same in all subclasses. Thus, it is desirable to give fields the same offset in subclasses as they have in their superclasses, so as to minimize the number of offset messages that must be dynamically-dispatched. We can preserve the offsets of fields easily from one of a class's parents. We make the simplifying assumption that fields are accessed with uniform frequency, and therefore we preserve the offsets of the parent class that has the largest number of fields. For example, suppose we have:

```
class C subclasses A, B { ... }
```

If class B defines 5 fields and class A defines 1 field, then our approach will lay out objects of class C with first the fields of B, then the fields of A, and then any fields that C defines (since this will keep the offsets of the most fields the same as they were in their parent classes). This strategy seems to work well in most cases, although we have observed situations where the single field defined on class A is much more heavily accessed than any of the fields on B. To address these situations, we are exploring the use of profile data to weight the ordering in which superclasses are laid out by the frequency with which fields in those superclasses are accessed.

There are many techniques for choosing offsets so as to minimize object size while keeping offsets the same for the largest number of objects. Existing approaches employ techniques such as inserting padding in objects [Dixon et al. 89] and using negative offsets [Pugh & Weddell 90, Myers 95], and they could also improve object layout in our environment.

Existing C++ implementations use a different approach for object layout in the presence of multiple inheritance. Whenever an object is to be viewed as one of its superclasses, the object pointer is adjusted to point to the beginning of that superclass's fields in the object [Stroustrup 87]. This moves the cost of handling multiple inheritance from the point of accessing a field to the point of using an object as one of its superclasses (and when sending a message, in some implementations), embodying a different set of tradeoffs than our approach. Virtual (non-replicating) inheritance also introduces additional costs in C++'s implementation. Under virtual inheritance, our approach introduces overhead in accessing an instance variable whose offset varies among its subclasses, while C++'s implementation of virtual inheritance imposes overhead on every access to an instance variable defined in a superclass and when sending a message. The pointer arithmetic used in C++'s implementation of multiple inheritance generates pointers to the middle of objects, which would introduce complexity in a garbage-collected environment like ours [Wilson 92].

4.5 Extending Standard Dataflow Optimizations to Analyze Instance Variables

As described in Section 4.2, it is often possible to optimize field accessor methods by converting them to direct field loads and stores which then become amenable to several dataflow optimizations. These optimizations can further reduce the overhead of implementing instance variables. For example, it is often not possible to evaluate field initializers eagerly with optimizations described in Section 4.3, so stores of the UNINITIALIZED value are performed as part of object creation. However, object creation is sometimes followed by explicit field initialization, in which case dead field store elimination discussed below can eliminate the unnecessary store of UNINITIALIZED.

4.5.1 Redundant Initialization Check Elimination

To implement the semantics of lazy field initialization, the reader accessors include a check for the UNINITIALIZED value. However, if a field is accessed multiple times, this initialization check is needed only for the first time, since any access causes the field to become initialized if it was not before. The subsequent checks are redundant and can be eliminated. This optimization can be implemented with a forward data-flow analysis by keeping track of which fields have been initialized (by either field loads or stores), and eliminating initialization checks for loads from these fields. Our compiler currently does not perform this optimization.

4.5.2 Dead Field Store Elimination

A dead field store, like a dead variable store, is one whose effects are never examined. For example, if there is another store to the same field later and no loads from the field between the two stores, the earlier store is dead and can be eliminated. This optimization, for example, often eliminates the stores of UNINITIALIZED that are later overwritten with real values.

Dead store elimination consists of two phases. First, it relies on accurate knowledge of the memory locations being accessed, and therefore needs alias analysis. Alias analysis is performed earlier with a forward pass.

Then, a backwards data-flow analysis pass detects which stores are dead. The analysis starts at the end of a procedure, where all memory locations could be considered live. To achieve better results, we perform an analysis to distinguish between memory locations that possibly escape the procedure, and memory locations that are never accessed outside it. Such “local” locations are dead at the end of the procedure. For a given statement in the procedure, if it is a field store, the corresponding memory location is added to the dead set; for a field load, the location is removed from that set. Stores to locations in the dead set are dead and can be eliminated.

4.5.3 Redundant Load and Store Elimination

It is possible to further optimize uses of fields which have been converted to direct loads and stores. For example, it is quite common to create an object, initialize it and then access some of its fields. This pattern results in redundant field loads. In general, a field load or store is redundant if it repeats the same load or store performed earlier or a load follows a store to the same field, and the field is not modified between the two operations. Elimination of redundant loads and stores eliminates unnecessary work for subsequent field accesses. It can even be done across procedure calls for fields that are declared immutable.

This optimization is a kind of common subexpression elimination and can be done with a forward data-flow analysis. In this case, for each load or store we need to remember the memory location being accessed in addition to the value being stored (for field stores). To support this we use the results of alias analysis which is performed for dead store elimination. Redundant load and store elimination is performed in parallel with alias analysis.

4.5.4 Dead Object Elimination

After the previous optimizations have been performed, in certain cases all stores to and loads from an object's fields have been eliminated as redundant or dead, and the only time the object is mentioned is when it is created. Then such an object is dead and its creation is eliminated. Dead objects are detected as part of dead assignment elimination with no special treatment.

5 Performance

In this section, we seek to characterize the costs of various aspects of our design under a variety of implementation strategies:

- What are the costs of accessing instance variables with dynamically-dispatched messages?
- What is the impact of supporting external role-based extensions?
- What costs are imposed by lazily-initialized fields?

To answer these questions, we examined several programs written in Cecil, ranging in size from 400 to 56,000 lines. Cecil is a pure object-oriented language with multi-methods that allows mixed statically and dynamically typed code, although from the compiler's point of view, the language is dynamically-typed. Our benchmark programs are described in Table 5.

Table 1: Cecil Benchmark Programs

Program	Lines[a]	Classes[a]	Fields[a]	Description
Richards	400	7	19	Operating systems simulation
Deltablue	650	9	12	Incremental constraint solver
InstrSched	2,400	36	55	MIPS global instruction scheduler
Vortex	56,000	968	1179	Optimizing compiler for object-oriented languages

a. Not including 11,500-line standard library, which includes an additional 208 classes and 83 fields.

The remainder of this section focuses on the performance of the InstrSched program. We have found that this application is reasonably predictive of the behavior of the larger applications, while the smaller benchmark programs are not [Dean et al. 95, Grove et al. 95]. Some partial data for richards, deltablue, and Vortex and all the raw data for InstrSched can be found in the appendix.

We compiled the benchmark programs using Vortex, our optimizing compiler for object-oriented languages. Vortex implements all of the optimizations described in section 4, including intraprocedural class analysis, class hierarchy analysis, and profile-guided receiver class prediction, as well as traditional compiler optimizations, such as common subexpression elimination, constant propagation, and dead assignment elimination [Aho et al. 86]. To gather bottom-line performance numbers, we ran the applications on a lightly loaded Sun SPARC-20/61 workstation.

5.1 Accessing Fields with Dynamically-Dispatched Messages

Table 5 reports application performance for the following optimization configurations:

- Base: This configuration utilizes hard-wired class prediction, intraprocedural class analysis, class hierarchy analysis, and profile-guided receiver class prediction to optimize non-accessor dispatches, but does not attempt to statically bind accessor-related dispatches.

- Base + Intra: Extends Base by allowing accessor-related dispatches to be optimized using intraprocedural analysis.
- Base + CHA: Extends Base by allowing accessor-related dispatches to be optimized using intraprocedural analysis and class hierarchy analysis.
- Base + Profile: Extends Base by allowing accessor-related dispatches to be optimized using intraprocedural analysis and profile-guided receiver class prediction.
- Base + CHA + Profile: All dispatches are eligible for full optimization.

Table 2: Impact of Fields as Messages

Configuration	Time (msec)	Space (KB)	Number of Dynamic Dispatches			Class Tests
			Non-Accessor Methods	Reader/ Writer Accessors	Offset Methods	
Base	3,850	2,093	416,346	1,196,733	1,196,733	2,257,198
Base+Intra			416,346	1,110,551	1,110,551	2,257,198
Base+CHA			416,346	265,208	221,313	2,242,758
Base+Profile			416,346	33,275	33,039	3,000,149
Base+CHA+Profile	2,340	1,943	416,022[a]	5,927	65,059	2,630,457

a. 324 additional non-accessor dispatches were eliminated due to downstream effects from eliminating accessor dispatches.

In addition to application runtime and compiled code space, Table 2 includes two more implementation-independent metrics: dynamic counts of dynamically-dispatched messages and class tests. The dispatch data is broken into three categories: invocations of non-accessor methods, reader and writer accessors, and offset methods. We chose a base configuration in which non-accessor messages were optimized "as well as possible" both to more closely approximate the performance impact of flexible instance variables on a less pure language than Cecil and to restrict the effects of the various optimizations to those messages introduced by fields. Vortex includes intraprocedural class analysis, class hierarchy analysis, and profile-guided receiver class prediction as part of a single combined optimization pass that eliminates dynamic dispatches. Since all of these analysis and transformations are occurring simultaneously with substantial interactions, and all three are enabled for non-accessor methods in all five configurations, it is quite difficult for us to directly and accurately measure the intermediate data points represented by the middle three lines of the table. Instead, the dynamic dispatch and class test data are derived from measurements of simpler configurations. For example, the data for Base+CHA was derived by measuring two otherwise unoptimized systems in which class hierarchy analysis was applied to either all messages or only non-accessor methods. The difference in the number of dispatches in these two systems was then combined with the dispatch data from the Base configuration to compute the final data. Thus, runtime and code space numbers are not shown for the intermediate configurations.*

* Appendix A contains direct measurements of the three intermediate configurations, but since we were unable to isolate the accessor and method enabled optimization from information derived from the other two method-only enabled optimizations, we believe that the indirect computation is more indicative of the relative strengths of the optimizations when applied in isolation.

As expected, intraprocedural class analysis by itself had little impact on accessor-related dynamic dispatches. Class hierarchy analysis was much more effective, eliminating 80% of the accessor related dispatches. Profile-guided receiver class prediction eliminated almost all the accessor-related dispatches, but increased the number of class tests by 30%. As expected, Base+CHA+Profile executed fewer class tests than Base+Profile, but it actually has more dynamically-dispatched offset methods. This non-intuitive result is an artifact of the interactions of the time/space tradeoffs made by our class-test-inserting heuristics and call-chain profile data [Grove et al. 95]. With class hierarchy analysis, more accessors are inlined, leading to longer call-chains and greater peakedness in the profile-derived class distributions for the offset messages, therefore fewer class tests are inserted than when using the flatter class distribution for the offset message from the non-inlined accessor method.

5.2 Impact of Externally-Defined Role Extensions

Allowing externally-defined role extensions has a significant impact on the implementation. In particular, since instance variables can be added to a class from other modules, the sizes and layout of objects cannot be determined until all role extensions are seen. Unless the compiler has a view of the whole program, this means that instance variable offsets and the sizes of objects cannot be determined until link-time. To examine the impact of this, we made two changes to our compiler for configurations lacking class hierarchy analysis:

- Since the sizes of objects are not known until the whole program becomes available, objects are allocated by calling a link-time-generated routine that allocates the appropriately-sized object and initializes its fields to UNINITIALIZED.
- Since fields can be defined externally, field offsets cannot be known until all the field declarations in the program are available. Under configurations without class hierarchy analysis, we simulate this effect by forcing all sends of offset messages to be dynamically-bound.

We modified each of the configurations that we measured in Table 5 to include support for roles. The results are shown in Table 3, where each of the configurations from the previous section is paired with a version that assumes external role-based extensions.

As expected, the lack of offset information imposes significant costs for configurations without class hierarchy analysis. The performance difference between Base and Base w/Roles reflects the cost of allocating objects using out-of-line routines, but this cost is only part of the impact of rules since neither of these configurations optimize accessor or offset methods at all. Although we are unable to measure the absolute execution times of the intermediate configurations, the number of dynamic dispatches due to the uncertainty of field offsets for these systems represents a substantial fraction of the total number of dynamic dispatches in the program for the more optimized configurations. However, an implementation of roles designed from scratch in a system that lacked class hierarchy analysis would likely use a cheaper form of resolving offsets than full-blown dynamic dispatches, such as determining offsets by loading them from a table [Harrison & Ossher 93]. Therefore we would be unable to draw strong performance conclusions from execution time data for such systems even were we able to measure them. Class hierarchy analysis is able to completely remove the impact of roles from the implementation, however, avoiding the need for clever ways of computing offsets at runtime.

5.3 Costs of Lazily-Initialized Fields

There are two primary costs for lazily-initialized fields in our implementation: the cost of storing the distinguished UNINITIALIZED value into an object's fields when the object is created, and the cost of checking for this value on any field access. Section 4.3 discussed two optimization techniques that can eliminate some of these stores: eagerly initializing fields, and eliminating UNINITIALIZED checks for fields that are known to always be initialized. In this section, we compare three configurations:

Table 3: Impact of External Role-Extensions

Configuration	Time (msec)	Space (KB)	Number of Dynamic Dispatches		
			Non-Accessor Methods	Reader/ Writer Accessors	Offset Methods
Base	3850	2,093	416,346	1,196,733	1,196,733
Base w/Roles	4060	2,069			1,196,733
Base+Intra			416,346	1,110,551	1,110,551
Base+Intra w/ Roles					1,196,733
Base+CHA			416,346	265,208	221,313
Base+CHA w/ Roles					
Base+Profile			416,346	33,275	33,039
Base+Profile w/ Roles					1,196,733
Base+CHA+Profile	2340	1,943	416,346	5,927	65,059
Base+CHA+Profile w/ Roles					

- Lazy, a configuration that uses class hierarchy analysis and profile-guided receiver class prediction to optimize the program, but does no optimizations directed at lazily-initialized fields.
- Lazy+Trivial, which adds the optimizations of eagerly evaluating trivial initializing expressions at object creation sites and eliminating the checks for UNINITIALIZED when accessing fields with trivial initializers.
- Lazy+Trivial+Explicit, which eliminates additional checks for UNINITIALIZED by examining all object creation points in the program to determine which fields are always initialized, either by the copying down of trivial initializers or by explicit user-supplied values at all creation points. This is the default optimization level for lazily-initialized fields in our system, and corresponds to Base+CHA+Profile in Tables 5 and 3.

Table 5 shows the results of our experiments. We present data in terms of absolute execution time and code space for our implementation, and also show the less implementation-dependent measures of the number of checks and stores of UNINITIALIZED remaining, and the total number of stores involved in initializing objects (as a point of reference for evaluating the number of stores of UNINITIALIZED).

Table 4: Impact of Lazily-Initialized Fields For InstSched Benchmark

Configuration	Time (msec)	Space (KB)	Checks for UNINITIALIZED	Stores of UNINITIALIZED	Initializing stores
Lazy	2740	1,984	1,020,280	38,291	96,833
Lazy+Trivial	2370	1,960	486,799	15,137	96,833
Lazy+Trivial+Explicit	2340	1,943	429,831	15,137	96,833

The trivial initializer optimization eliminated slightly more than half of the checks for UNINITIALIZED, which make up the bulk of the cost of our lazily-initialized field implementation. Employing whole-program analysis to examine all of the object creation points in the program, in addition to all the field declarations, was able to eliminate an additional 10% of the remaining checks. The bottom-line execution performance improved by approximately 20% due to our optimizations, and we hypothesize that the remaining checks for UNINITIALIZED imply that the remaining cost of lazy initialization represents less than 20% in bottom line performance in our system.

6 Conclusions

We have presented a semantics for instance variables that supports a strong decoupling between a class and its clients, its subclasses, and its role extensions, and have discussed implementation techniques and tradeoffs for our semantics. For our design, we combined several features found separately in other languages, such as accessor methods, immutable instance variables, default initialization expressions, and atomic initialized object creation. We also added several new features, such as lazily initialized fields and allowing references to `self` in intialization expressions, which are especially important for role-based programming.

We also examine implementation issues for our semantics, showing how existing compiler techniques, such as class hierarchy analysis, can dramatically reduce the cost of flexible instance variables, as well as how several new optimization techniques reduce the cost of lazily-initialized fields. To assess the impact of these compiler optimizations, we presented experimental results for a variety of optimization combinations.

Several issues remain unresolved and we hope to address these in future work. First, atomic object creation makes it difficult to inherit constructor code and to define instance variables in external roles without having to modify any `copy` methods defined for the class; we are working on extensions to our semantics to resolve these issues. Second, the performance impact of our language features on lower-level, hybrid object-oriented languages, such as C++, remains an open question, and we are adapting a C++ language front-end onto our Vortex compiler's optimizing back-end to address this question.

Our design and implementation has been used in practice by several people developing a large application over several years. The features of our instance variables have helped to make writing and evolving this application significantly easier.

References

[Aho et al. 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[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.

[Chambers & Leavens 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. In *Proceedings of the 1994 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Portland, OR, October 1994.

[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. Conference on Programming Language Design and Implementation.

[Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993. Revised, March 1997.

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

[Dean et al. 95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 1995 European Conference on Object-Oriented Programming*, LNCS 952, Aarhus, Denmark, August 1995. Springer-Verlag.

[Dixon et al. 89] R. Dixon, T. McKee, M. Vaughan, and Paul Schweizer. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–214, New Orleans, LA, October 1989.

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

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

[Grove et al. 95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP95], pages 108–123.

[Harrison & Ossher 93] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of the 1993 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, Washington, D.C., October 1993.

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

[Hudak & Young 86] Paul Hudak and Jonathan Young. Higher-Order Strictness Analysis in Untyped Lambda Calculus. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 97–109, January 1986.

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

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

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

[OOP95] *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages,*

and Applications, Austin, TX, October 1995.

[Ossher et al. 95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-Oriented Composition Rules. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP95], pages 235–250.

[Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Pugh & Weddell 90] William Pugh and Grant Weddell. Two-Directional Record Layout for Multiple Inheritance. *SIGPLAN Notices*, 25(6):85–91, June 1990. Conference on Programming Language Design and Implementation.

[Reenskaug & Anderson 92] Trygve Reenskaug and Egil P. Anderson. System Design by Composing Structures of Interacting Objects. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 133–152, 1992.

[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 of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 9–16, Portland, OR, November 1986.

[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.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Orlando, FL, December 1987.

[Ungar 95] David Ungar. Annotating Objects for Transport to Other Worlds. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* [OOP95], pages 73–87.

[VanHilst & Notkin 96] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *Proceedings of 2nd International Symposium on Object Technologies for Advanced Software*, March 1996.

[WB & Johnson 90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

[Wilson 92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*, LNCS 637, St. Malo, France, September 1992. Springer-Verlag.

Appendix A Appendix A

This appendix presents the raw data used to create Table 5. Optimizations with the NA suffix were enabled for non-accessors methods only.

Table 5: Raw Data

Applications	Configuration	Time (sec)	Space (MB)	Number of Dynamic Dispatches			Class Tests
				Non-accessor Methods	Reader/ Writer Accessors	Offset Methods	
Richards	unopt	11.16	1.03	7,504,692	2,500,710	2,500,710	0
	intraNA	2.13	0.94	1,196,975	2,500,710	2,500,710	2,208,615
	intra	2.20	0.92	1,196,975	2,500,502	2,500,502	2,208,615
	intraNA+CHANA	1.98	0.92	891,923	2,500,710	2500710	2,208,613
	intra+CHA	0.61	0.86	891,923	691,490	0	2,189,997
	intraNA+ProfileNA	1.68	0.95	810,985	2,500,710	2500710	2,281,598
	intra+Profile	0.52	0.95	810,985	155,068	139170	4,684,296
	intraNA+CHANA+ProfileNA	1.64	0.92	609,637	2,500,710	2500710	2,281,595
	intra+CHA+Profile	0.35	0.87	609,637	9,349	0	3,036,390
Deltablue	unopt	2.49	1.04	862936	185650	185650	0
	intraNA	0.75	0.94	370178	185650	185650	232,535
	intra	0.76	0.94	370178	181832	181832	232,535
	intraNA+CHANA	0.76	0.93	267382	185650	185650	206,933
	intra+CHA	0.48	0.88	267272	84002	0	206,733
	intraNA+ProfileNA	0.35	1.01	33634	185650	185650	241,653
	intra+Profile	0.16	0.99	33624	14123	8338	323,077
	intraNA+CHANA+ProfileNA	0.38	0.96	18190	185650	185650	220,629
	intra+CHA+Profile	0.11	0.93	18180	3619	0	274,514
InstrSched	unopt	17.83	2.22	7531918	1196733	1196733	0
	intraNA	8.64	2.01	3487717	1196733	1196733	2,412,677
	intra	8.69	1.97	3487717	1110551	1110551	2,412,677
	intraNA+CHANA	7.46	1.99	2480813	1196733	1196733	2,152,275
	intra+CHA	5.97	1.84	2480504	265208	221313	2,137,835
	intraNA+ProfileNA	3.77	2.10	438443	1196733	1196733	2,290,288
	intra+Profile	2.50	2.08	438228	33275	33039	3,033,239
	intraNA+CHANA+ProfileNA	3.85	2.09	416346	1196733	1196733	2,257,198
	intra+CHA+Profile	2.34	1.94	416022	5927	65059	2,630,457

Table 5: Raw Data

Applications	Configuration	Time (sec)	Space (MB)	Number of Dynamic Dispatches			Class Tests
				Non-accessor Methods	Reader/ Writer Accessors	Offset Methods	
Vortex	unopt						
	intraNA						
	intra						
	intraNA+CHANA	995	10.49	136236303	94753798	95252308	81263399
	intra+CHA	755	9.05	135848009	22375394	2818909	78324074
	intraNA+ProfileNA						
	intra+Profile						
	intraNA+CHANA+ProfileNA						
	intra+CHA+Profile						