Fast Object Operations in a Persistent Programming System

by

Andrew C. Myers

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1994

Author ....
Department of Electrical Engineering and Computer Science
April 15, 1994

Certified by ....
Barbara Liskov
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Committee on Graduate Students

Fast Object Operations in a Persistent Programming System

by

Andrew C. Myers

Submitted to the Department of Electrical Engineering and Computer Science
on April 15, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Object-oriented, persistent programming languages offer a simple model for the programmer to write applications that share data, even among heterogeneous systems. However, poor performance limits their general acceptance.

I present two independent but complementary techniques to speed up object operations in Theta, a object-oriented, persistent programming language that is used within the object-oriented database Thor. I examine the overhead that such a system imposes on operations such as method calls and field accesses, and show how these operations can be made fast without sacrificing the flexibility and extensibility of Theta.

First, I describe a novel object layout that allows method dispatch as fast or faster than C++ implementations, while keeping the fast field accesses associated with non-object-oriented systems. This layout derives from insights about the separation of subtyping and inheritance.

Second, I show how customization can be applied to Theta to avoid many method dispatches. The statically-typed nature of Theta makes it possible to avoid dispatches more effectively than in a dynamic system such as SELF. Dispatches can also be avoided in a novel use of customization that depends on the set of objects actively being used by an application, and which is applicable to persistent object systems.

Thesis Supervisor: Barbara Liskov
Title: NEC Professor of Software Science and Engineering

Acknowledgments

I thank my advisor, Barbara Liskov, whose insightful ideas and suggestions helped me to develop this thesis. The presentation was also greatly improved by her proofreading and stylistic comments. Working with her is a real pleasure.

My officemates, Mark Day and Umesh Maheshwari, have been great companions and sources of fun. I thank Mark especially for all his careful readings of this thesis, his very constructive criticisms, and his always-useful advice on how to navigate the shoals of graduate school.

Sanjay Ghemawat and Dawson Engler often pushed me to finish my thesis but compensated by slowing me down with workouts. Jim O'Toole has been a terrific housemate and an endless source of good ideas and intriguing discussions. All these people, plus others such as Wilson Hsieh, Franklyn Turbak, and Eric Brewer, provided helpful feedback on how to present this work.

I thank Dennis Allison, David Ciemiewicz, Kipp Hickman, Bruce Karsh, and Rob Myers, who contributed to making computers sufficiently interesting that I went back to school to study them.

Finally, my family has been very supportive of my goals and efforts. I especially thank my parents for the many things they've done to make this possible.

Contents

Chapter 1

Introduction

Persistent programming languages are found at the heart of object-oriented databases and interpreted programming environments. They conveniently bridge the gap between primary and secondary storage of data objects and provide a simple model for writing applications that share data. Unfortunately, most persistent programming languages do not perform as well as statically-compiled, non-persistent systems, limiting their general acceptance.

Because persistent objects may exist for a long time, performance decisions may need to be revisited. In programming languages like C++ [12] or Modula-3 [2], programmers may increase performance by limiting the extensibility of programs — for example, by exposing some implementation details of the types used. These limitations are unacceptable in an evolving, persistent system. The goal of this thesis is to show how good performance can be achieved without sacrificing extensibility.

In this thesis, I look at the distributed, object-oriented database Thor, and show how the performance of its programming language, Theta, can be made comparable to that provided by ordinary statically-compiled programming languages such as C or C++. Since almost all computation in Theta is composed of a few primitive object operations (for example, object method invocation), I focus on ways to make these object operations as fast as possible.

In the rest of this chapter, I present some necessary background on Thor, particularly on the object-oriented, persistent programming language Theta (Since both Thor and Theta are still being designed at the time of this writing, they are likely to differ in small ways from what is described here). I then explain the set of object operations that I have worked to optimize.

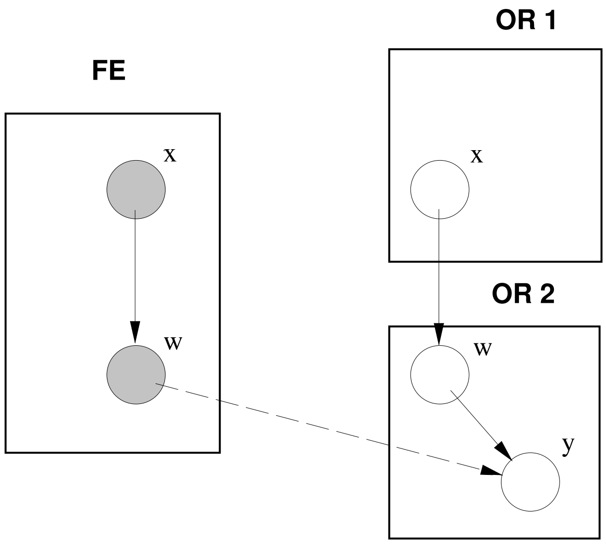

Figure 1-1: Structure of Thor

1.1 A Sketch of Thor

Thor provides persistent storage for a large number of objects that are stored in reliable servers called *object repositories* (OR's). An object has encapsulated state and is accessed only through a set of *methods*. The object state may contain references to other Thor objects, even those located in other repositories.

Thor is based on a client-server model. The servers only store objects; computation is performed on persistent objects at the clients, which are called *front ends* (FE's). The FE is a piece of Thor that executes on the client machine. A user application can exist partly outside Thor, on the same client machine as the FE, but for safety all method calls on Thor objects execute within the FE.

As an application executes, the FE *fetches* copies of those persistent objects that it needs from the repositories (Figure 1-1). In this figure, object copies at the FE are represented by gray circles; white circles represent objects stored persistently at a repository. An object copy is fetched when a reference is followed from an object at the FE to an object that is still only at an OR, *e.g.* object y in the figure.

When computation completes at a front end, changes to the persistent objects are sent back to the repositories to be stored permanently. Computation at an FE is divided into a series of *transactions*. A transaction can perform an arbitrary amount of computation, but its modifications to object state are *committed* atomically to the repositories.

This thesis is primarily concerned with making code run fast, and so it focuses almost exclusively on what happens at a front end.

1.2 Theta Language Features

The Theta programming language is used to describe computations that occur at the FE. Theta is used to specify object methods, to describe the state encapsulated by objects, and to implement methods using that state.

Theta is statically typed, because static typing aids in building reliable systems; however, this type information also helps in making the optimizations described here. The limitations of static typing are ameliorated in Theta through parametric polymorphism and more expensive mechanisms like casting down and class evolution — topics beyond the scope of this thesis.

At the time of this writing, Theta has not been completely specified; however, a reference manual is forthcoming [9].

1.2.1 Types and Classes

Theta is an object-oriented language, with multiple supertypes and single inheritance. A type specifies an *interface*: a set of methods that can be performed on objects of that type. However, types do not specify the implementations of those methods. In fact, a type may have *multiple implementations*, or *classes* that implement it. In this thesis, the terms *implementation* and *class* are used interchangeably.

Theta supports the ability to add new interfaces or implementations incrementally, so that entirely new kinds of objects can continue to be added to the system.

Types exist in a *type hierarchy.* Each type has zero or more *supertypes*, creating a directed acyclic graph. A subtype supports all the methods of its supertypes, and extends their interfaces by adding new methods. An object of type T can be accessed through the interfaces of any of the supertypes of T. The ability to have multiple supertypes is not unusual, although many object-oriented programming languages restrict types to having a single supertype.

One example of a type is `rotation`, which represents a rotation in ordinary three-dimensional space. A `rotation` has four methods, named `compose`, `apply`, `axis`, and `angle`. An informal specification is provided for each method of the type in the Theta declaration that follows.

```
rotation = type
    % A "rotation" represents a rotation in 3-space
    compose(r: rotation) returns(rotation)
        % Return the "rotation" that is the composition of this rotation
        % with "r".
    apply(p: point) returns(point)
        % Rotate "p" by this rotation and return the result.
    axis() returns(point)
        % Return a normalized vector that is the axis of the rotation.
    angle() returns(float)
        % Return the amount of the rotation, which is a number in
        % [-pi, pi).
end
```

In addition to the arguments listed in the specifications above, all these methods have an implicit argument named `self`, whose type is `rotation`. The `apply` method, for example, applies the rotation `self` to the point `p`, producing a new point. The argument `self` is called the method's *receiver*. In most object-oriented languages, the receiver is denoted by a special variable within the code of a method, such as `self` or `this`.

A class is an implementation of a type. A class provides code to implement the methods of its type, and defines object fields that the code manipulates. In addition to providing the methods of the implemented type, a class may have additional *private* methods that are only used internally.

There are several ways to write a class that implements `rotation`. One possible implementation of `rotation` looks like:

```
quaternion = class rotation
   r,i,j,k: float

   ...

   angle() returns(float)
       return 2.0 * arccos(self.r)
   end angle

   maker xrot(float angle)
       init { r := cos(angle / 2.0),
              i := sin(angle / 2.0),
              j := 0,
              k := 0 }
   end xrot
end quaternion
```

where all the interesting aspects of the quaternion implementation have vanished into the "`...`". Note that objects of the class `quaternion` have four fields, or *instance variables*, which can be accessed in the methods of these objects. For example, the `angle` implementation accesses the `r` field.

The maker (or constructor) `xrot` is not defined in the type, because it is not a method of objects of class `quaternion`. Calling a constructor is much like calling a procedure to create a new object. An example of a constructor is presented here only to illustrate where objects come from in the first place.

In Theta, objects are only manipulated through calls to methods that are described by a type interface, such as `rotation`. This abstraction barrier allows the interface to be reimplemented by a new class or classes, without breaking any code that is a client of the interface. Some object-oriented languages, such as C++, allow more information about a class, such as its instance variables, to be exported to the outside world. When instance variables are used by clients, the clients become dependent on the internal structure of objects. New implementations of the interface cannot be created unless they have the same instance variables, because otherwise the client code would break.

For example, if the quaternion implementation served as the rotation abstraction, new implementations of rotations would be forced to have the four instance variables of `quaternion`, because client programs might access these instance variables directly and would otherwise break.

1.2.2 Method Dispatch

Let us consider what it means to invoke an object method in a type hierarchy, where a given type may have multiple implementations. Given an object x of declared type T, the compiler cannot statically determine the proper code to run when a method is invoked; although x is declared to be of type T, the implementation of x may be any of the potentially many implementations of T or of some subtype of T. Each implementation may have its own code for each method described by T. Knowing the declared type of x does not provide precise information about the correct code to run.

Instead, the decision about what code to run must be made at runtime. This runtime decision is called *dynamic dispatch*, and it is the source of both the power of object-oriented languages and of the loss of performance often associated with them. Dispatching is the operation that fundamentally distinguishes object-oriented languages from strictly procedural languages.

In the following example, the call to the method `apply` performs a dynamic dispatch on an object of type `rotation`. In general, the compiler cannot determine which of the two possible implementations (`quaternion` and `rotmatrix`) the variable `x` refers to. Only at runtime can the system make the determination.

```
x: rotation
...
if (complex-condition) then
   x := quaternion.xrot(pi/2)
else
   x := rotmatrix.make_rot('x', pi/2)
end
p2: point := x.apply(p1)
```

Note that within one execution of this program, the variable `x` can refer to objects of several different implementations.

Dynamic dispatch is important because it provides extensibility. A programmer can write code using an existing interface (a type), intending to use the code with a particular implementation of the interface; yet, later the implementation of that interface can be changed. In fact, multiple implementations of that interface can be used in the same piece of code.

Dynamic dispatch requires that the correct implementation procedure be looked up for the object on which the method is being invoked. Programming languages use different approaches to speeding up this lookup, but dispatch is always more expensive than a procedure call.

Some conventional programming languages, such as C++, allow the programmer to annotate methods to indicate whether they should be subject to dynamic dispatch semantics. These *non-virtual methods* are only as expensive to call as a procedure because the correct implementation can be determined at compile time, based on the static type of the object. Non-virtual methods can even be inlined in C++, although the programmer is again required to indicate this explicitly.

The use of non-virtual and inlined non-virtual methods in C++ can provide a significant performance improvement. For example, the Richards benchmark[4] runs 50% faster when inlined non-virtual methods are used. The data from this simple experiment, which was performed on a DEC AXP 3000/400 system, are shown in Figure 1-2. In both cases, the benchmark was compiled by the DEC C++ compiler

Virtual methods	201 ms
Non-virtual methods	104 ms

Figure 1-2: Richards benchmark times with and without method dispatching

`cxx`, with optimization.

These data suggest that eliminating some of the overhead of method dispatch can result in significant performance improvements. However, non-virtual methods are not a good way of eliminating this overhead; they restrict the set of valid extensions of a type. If a new implementation of the type is added, the non-virtual method will no longer work properly, because static type information will not be sufficient to determine the correct method implementation. The effect of this restriction on new implementations is that the programmer is forced to trade off extensibility for performance.

Trading off extensibility for performance is a trademark of the C++ philosophy, but it is not suitable for a persistent object database, where assumptions about needed functionality may become invalid as the system evolves. Because the system is long-lived, extensibility must always come first; we must look for ways to achieve good performance without sacrificing extensibility.

1.2.3 Inheritance

Theta supports *inheritance*, a feature which allows one class to inherit code from another class, called its *superclass*. Rather than implement a method in its type interface, it can simply use the superclass implementation. The subclass object inherits all of the fields of superclass objects, since otherwise the inherited methods might not make sense. The subclass may also add more fields and override methods. When a subclass overrides a method, this affects the behavior of other inherited methods. If another inherited method calls the overridden method, it will invoke the subclass implementation of the method rather than the superclass implementation.

A subclass need not implement a type that is related to the type that the superclass implements. The superclass methods that are inherited might not even appear in the interface of the subclass's type. In this case, inheritance is being used purely as a implementation shortcut, though a very useful one.

Note that subtyping and inheritance differ in Theta. Subtyping is generally considered to be the essential contribution of object-oriented programming. In a system with subtyping, a single object can satisfy interfaces presented by multiple types — a powerful form of polymorphism. Inheritance, on the other hand, is a useful mechanism for producing classes that implement these interfaces.

The class inheritance mechanism in most object-oriented languages is very powerful. Two distinct major uses of inheritance have been identified [1]: *interface inheritance*, and *implementation inheritance*. Interface inheritance is identical to subtyping – that is, interface A inherits from interface B if the type A is a subtype of B. Thus, interface inheritance is a relation between types. The C++ facility of *abstract superclasses*, allowing a class to inherit only the interface of its superclass, is an example of interface inheritance.

Implementation inheritance is the ability to implement one interface by reusing code that was written for a different interface. Implementation inheritance is a relation between classes. However, implementations often are inherited from a class that implements a related type.

Implementation inheritance has two major benefits: programmer convenience and code sharing.

Inheriting an implementation is convenient because it means that code does not have to be rewritten for a subclass when it would be mostly identical to the superclass code. This situation is quite common in object-oriented systems, especially when the subclass and superclass implement interfaces that are in a subtype relationship.

Implementation inheritance also reduces the total amount of code in the system, since multiple classes can use the same piece of code. This reduction in total code size can result in better code cache and virtual memory performance. However, code sharing is only a performance optimization. Inheritance can still provide programmer convenience if the inherited code is recompiled for the inheriting class.

From this point forward, I will use the term “inheritance” to mean “code-sharing implementation inheritance,” and the term “supertype” to mean “the type from which the interface is inherited.”

1.3 Theta Object Operations

Computation in Theta consists almost entirely of a series of primitive object operations. There are four different operations that user code can perform on objects, and an operation that the garbage collector performs on objects. Making Theta run fast is largely a matter of making these primitive object operations run fast individually.

In addition to methods, Theta objects also have *fields*, sometimes called *instance variables* or *member variables*. A field is a mutable slot in the object that can be accessed by object methods, but not from outside the object's implementation.

Most of the computation in Theta consists of four ordinary object operations:

- Executing an object method (dispatching)

  Example:
  ```
  x: rotation
  ...
  x.apply(p1)
  ```

- Reading from an object field, which is allowed only within the implementation of that object's class.

  Example: `y: float := x.r`

- Writing to an object field – also allowed only within class implementations.

  Example: `x.j := 0`

- Viewing an object by a different interface than the current one: *casting up* the type hierarchy. In the example, the type `any` is the universal supertype.

  Example: `z: any := x`

Reading or writing object fields is exactly like reading or writing fields in records of ordinary programming languages, except that the object's implementation may have inherited some fields from a superclass.

Casting up is an operation that is usually invoked implicitly, when an object is passed as a formal parameter to a method expecting a type different from the known type of the object. In object-oriented systems with only single inheritance, casting up is usually a free operation. In a system with multiple inheritance and method renaming, casting up must have some cost, as it changes the interface though which the object is viewed. Implementations of C++ have shown that allowing this operation to have a one-instruction cost can yield significant speedup in dispatching

[24]. An object reference can be represented as any of a number of distinct memory pointers, depending on the declared type of the object. Casting up can be performed by offsetting a pointer by a fixed constant. The details of this scheme will be examined in Chapter 2.

A fifth object operation is provided by some object-oriented programming languages, such as Trellis/Owl [22, 23] and Modula-3 [18]. This is the operation of *casting down* the type hierarchy, sometimes provided as a *typecase*, and sometimes as a simple cast that is allowed to fail when the object is not of the desired type. Although casting down is an important operation in some programming styles, it is usually infrequent. Optimizing such casts is a separate problem, and beyond the scope of this thesis.

The garbage collector implicitly performs a kind of casting-down operation. Given an object pointer, the garbage collector must be able to locate the beginning of the object and the fields of the object. Although it only affects garbage collection, this operation must also be supported efficiently.

1.4 Outline of the Thesis

In this thesis, I look at the basic object operations of method dispatching and accessing fields, to see how they can be made efficient. Since almost all computation within Theta is composed of these primitive operations, reducing the amount of time spent per operation is very important.

The expected amount of time per operation is on the order of a few machine instructions. Thus, removing even a single instruction from the standard instruction sequence for method dispatch can have a noticeable impact on overall system performance.

Because I measure time in machine instructions, it is worthwhile to discuss what machine is under consideration. I assume a fairly standard RISC architecture, and use the MIPS R3000 as an example of such an architecture. Most current RISC architectures do not differ significantly with respect to the instructions used here, so the particular choice of RISC architecture is not very significant.

My approach to making Theta run fast has been twofold. First, I have investigated how objects can be structured so as to minimize the length of the most time-consuming code needed to perform the basic operations. I describe the object layout I have designed in Chapter 2.

Second, I have examined the object operations and looked for special cases that turn out to be common in practice, that can be identified statically by a compiler, and that can be implemented more efficiently by the compiler than the general case. Such special cases can be exploited to produce faster code overall. I describe this compiler support in Chapter 3.

In Chapter 4, I summarize my design and its benefits, and describe some of the ways that this work could be extended or improved.

Chapter 2

Making Dispatch Fast

2.1 Overview

This chapter addresses the problem of making object operations — particularly, method dispatch — fast in those cases when it is unavoidable. The next chapter looks at ways to avoid method dispatch when it is not really needed; however, in a program that uses method dispatch heavily, making it fast is critical.

A great deal of work has been done to make method dispatch fast in various object-oriented languages. The object layout that I have developed for Theta is based on the object layout and dispatch mechanism of C++. Both of these mechanisms depend on the static typing of their respective languages. For this reason, I begin by discussing dispatch tables, a technique common to most fast dispatch mechanisms. I then cover the C++ dispatch mechanism, which is an refinement of the simple dispatch-table approach for supporting multiple inheritance.

In Section 2.5, I explain the *bidirectional object layout* that I have developed for Theta, and how it refines the C++ object layout for a persistent object system with separate inheritance and supertype hierarchies. It provides slightly faster method dispatch and a more compact object layout.

Finally, in Section 2.6, I discuss the current implementation of the Theta dispatch mechanism and provide some performance results.

2.2 Issues

At the same time that method dispatch is made fast, we must consider the impact on the other object operations described in section 1.3: accessing fields, locating the beginning of the object, and casting the object up the type hierarchy.

Before discussing my approach to object layout, and various approaches that lead up to it, it is worthwhile to articulate some of the assumptions I use in analyzing the performance of a solution.

2.2.1 Memory Usage

Memory usage is an issue for object layouts. Most fast dispatching schemes share a table or set of tables among all objects of a class. Some researchers have been concerned about making these tables as small as possible [11, 20]. However, since dispatch tables are shared, they consume a small fraction of memory, assuming that there are a reasonable number of objects per dispatch table. This is a good assumption in class-based object-oriented languages, although it may be untrue for systems such as SELF, where individual objects can rewrite their method implementations. I do not place a high priority on minimizing per-class memory usage, but minimizing the amount of *per-object* overhead is very important.

2.2.2 Instruction Sequences

Analyzing the length of instruction sequences for performing method dispatch is tricky. The usual conceptual model of RISC architectures is that each instruction takes one cycle. However, certain instructions can result in pipeline stalls, effectively increasing the duration of the instruction.

For example, any reasonable dispatch sequence must contain a jump to an address specified by a register. This instruction may cause pipeline stalls, because the instruction-fetching unit of the processor is not able to determine the target of the jump. Jump instructions can also cause instruction cache misses for similar reasons. However, since all dispatch sequences considered in this chapter contain this instruction, I ignore the effect of the jump instruction for purposes of comparison.

Another problematic instruction is a load from memory, which can generate a data cache miss and subsequent stall while the memory subsystem obtains the data value from a secondary cache or main memory. For this reason, I separate loads into two classes: loads that are expected not to generate a cache miss, because they access a frequently-used location; and loads that are likely to generate a cache miss. Loads that hit in the cache cost only a single cycle. Loads generating a cache miss can be expected to take significantly longer than one cycle, although how much longer depends on details of the architecture.

2.2.3 Statically-Typed vs. Dynamically-Typed

Object-oriented programming languages are divided into two major camps: statically typed vs. dynamically typed. In the former camp are languages such as C++ [12], Modula-3 [18], and Trellis/Owl [22]. In the latter are Smalltalk [13], SELF [5], and Objective C [8]. Dynamically-typed languages provide greater polymorphism, since there is no type system to limit the kind of object that can be provided as a method argument. However, these languages do not have the advantages of compile-time type-checking.

Performing method dispatch is a more complex process in dynamically-typed languages. In a language like Smalltalk or SELF, there is no guarantee that an object will even understand a method called on it. The system must always be ready to detect an invalid method call and generate an appropriate exception or error — a capability with some runtime cost.

An invalid method call cannot occur in statically-typed languages, as the type system guarantees that the object will understand the method being called on it. Static type-checking, as in Theta, simplifies the problem of performing method dispatch quickly.

2.3 Dispatch Tables

The *dispatch table* is a basic mechanism that is used in some form by many language implementations. It was originally used for statically-typed programming languages with single inheritance, such as early versions of C++. Dispatch tables are discussed here in the context of C++ in order to facilitate the next section, in which more advanced C++ dispatch mechanisms are covered. This discussion applies to any statically-typed language with single inheritance.

Early versions of C++, such as the AT&T C++ 1.2 compiler, allowed only single inheritance and single supertypes, making possible a particularly simple form of dispatch. Each method of a type is assigned a unique small integer index (Note that in C++, types and classes are identical). The root classes number their methods consecutively, starting from zero, and a subclass's lowest-numbered additional method has an index one greater than its superclass's highest. Thus, an object of class C supports a set of n methods, each of which has a unique method index in the range $0 \ldots n-1$. For each of these methods, its index in the superclass of C is the same as in C. Applying this rule transitively, we can see that the method's index is the same all the way to the top of the type hierarchy.

For example, consider the following type specifications. For uniformity, all type specifications in this document will be written in Theta syntax, even when C++ is under discussion.

```
point = type
   x() returns(int)
   y() returns(int)
end point

c_point = type supers point
   get_color() returns(color)
   set_color(c: color)
end c_point
```

A `point` only supports the `x` and `y` methods. Because `c_point` is a subtype of `point`, it supports `x` and `y`. It also has an associated color that can be observed or changed by `get_color` and `set_color`. Since `point` has no supertypes, its methods will be assigned the indices $\mathtt{x} = 0, \mathtt{y} = 1$. Objects of type `c_point` must agree with `point` objects about the indices of `x` and `y`, so `get_color` $= 2$ and `set_color` $= 3$.

Each class has a *dispatch table* that is shared by objects of that class. The objects start with a one-word header that points to this shared dispatch table (Figure 2-1). Objects also have a non-shared portion, in which the fields of the object are stored. In the figure, this portion of the object is shaded in gray.

A subclass must conform to the layout of its superclass so that inherited methods can work properly. Only limited changes to the dispatch table format are allowed — specifically, the dispatch table may only be extended in order to add new methods. Note that in Figure 2-1, the dispatch table for `c_point` starts out with the same methods as `point`.

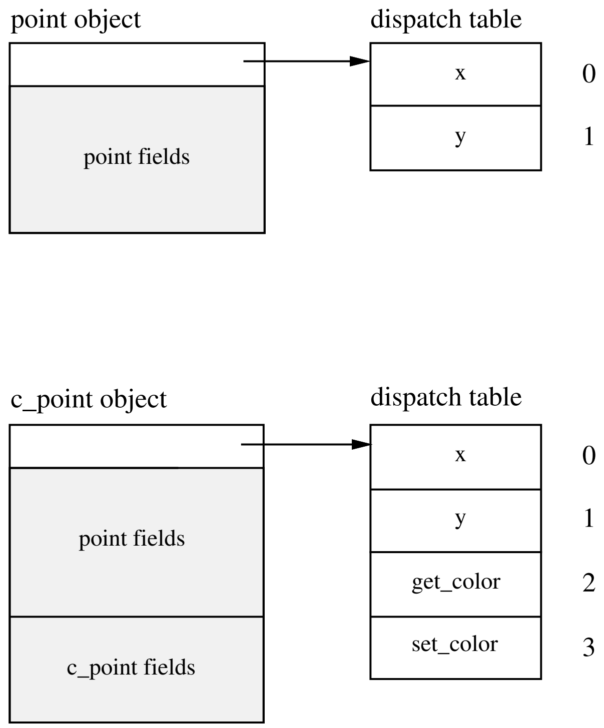

Figure 2-1: Objects and dispatch tables

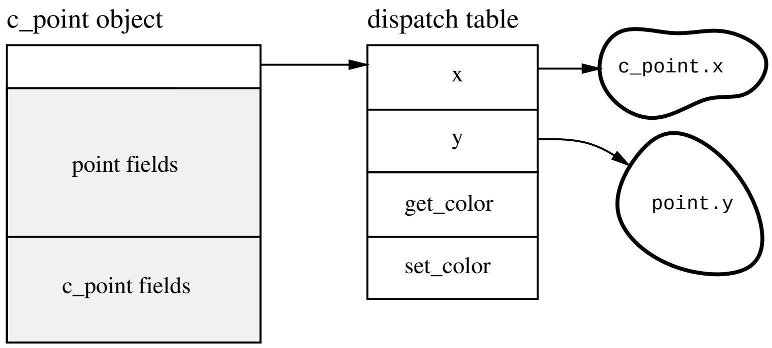

Figure 2-2: Dispatch tables and code

```
load t1, (a0)        % load the dispatch table address
load t2, i(t1)       % load the method code pointer
jalr t2, ra          % jump to the method
```

Figure 2-3: MIPS code for a simple dispatch

As shown in Figure 2-2, an entry in a class's dispatch table contains a pointer to the class's code for the method of that index, represented by an irregular balloon-shape. The notation $C.M$ denotes class C's implementation of method M.

A subclass may inherit method implementations directly from the superclass. In this simple dispatch-table model, inheritance is implemented simply by making the dispatch table for the subclass point to the code for the superclass methods. For example, in Figure 2-2, the `y` implementation from `point` is inherited by `c_point`, whereas the `x` implementation is not. Of course, a subclass can always override the code pointers in its dispatch table to point to code different from the superclass's. The subclass then provides a different implementation of those methods.

From here on, the actual code implementing a method will not be illustrated in the figures. When applicable, the $C.M$ notation will be placed in the dispatch table slots to indicate which code is being pointed to.

In order for inherited methods to work properly, the fields of the subclass must also be compatible with the superclass fields. The inherited code will expect to find fields at particular locations in the object, so the fields must also be inherited. Therefore, the subclass must retain all the fields of the superclass. It may also add new fields to the end of the object. An object of the subclass looks exactly like an object of the superclass as far as any inherited superclass method is concerned. For example, the `c_point` object begins with `point` fields, but adds some fields of its own to the end. So any method inherited from `point` will find that the `c_point` object exactly meets its expectations.

When method i is invoked on an object, the address of the dispatch table is loaded from the header of the object. The address of the code for method i is then loaded from index i in the dispatch table, and control is transferred to that loaded address. The assembly language describing this process is shown in Figure 2-3. The code assumes that register `a0` contains the object pointer, and `t1` and `t2` are temporary registers.

In some later figures, the dispatch tables will be elided, and implied through the

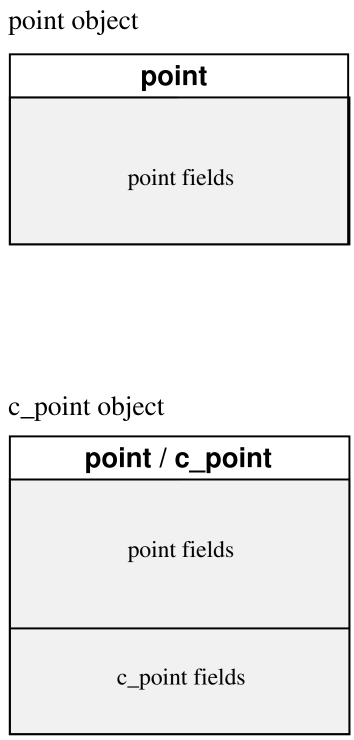

Figure 2-4: A more compact notation

more compact notation shown in Figure 2-4. Any slots in the object that point to dispatch tables are indicated by writing a sequence of types or classes, like A / B / C, in the slot. This notation indicates that the slot points to a dispatch table containing pointers to methods of the types or classes A, B, and C. Additionally, it implies that the method indices of A start at 0; that B's methods follow sequentially after A's; and similarly, C's follow B's.

For single-superclass systems, this dispatching and inheritance technique works extremely well, as a dispatch costs only three instructions. Field accesses require only a single load or store, just as in non-object-oriented systems. And casting up is a free operation. The limitation of this approach, addressed by newer C++ implementations, is that classes can have only a single superclass.

2.4 Multiple Superclasses in C++

The technique described above for implementing dispatch in a single-superclass system does not work for systems where a type can have multiple supertypes. In modern versions of C++ with multiple inheritance, the simple dispatch table is not sufficient. Consider what happens when a new class with multiple superclasses is introduced to the system. Each of the superclasses has its own mapping from method indices to methods, and they may use the same method index to refer to different methods. Unless the rule for assigning method indices is changed, the method indices of the superclasses may conflict. The dispatch tables of the superclasses are not, in general, compatible. Simple method indices cannot be assigned in a non-conflicting way in a system of separate compilation, assuming that a class implementation is compiled using only information about classes above it in the hierarchy.

Collisions could be made unlikely by assigning method indices from a very large space, but this approach would result in very large dispatch tables and inefficient use of memory. Dense assignment of method indices is desirable in order to make dispatch tables compact.

2.4.1 Embedded Objects and Pointer Offsets

The dispatch table technique can be extended to solve the problem of colliding method indices by using multiple dispatch tables [24]. This is the technique used in the AT&T C++ 2.0 compiler and other C++ compilers. The basic idea is to embed superclass objects within the subclass format. As described earlier, in a single-superclass system, the superclass object can be embedded at the beginning of the subclass, and the two dispatch tables can be merged into a single dispatch table in which the superclass's methods come first.

Additional superclasses can be placed sequentially after the subclass information (Figure 2-5). In the figure, D is a class with three superclasses: A, B, and C. The fields for the superclasses B and C are embedded after the fields for D, whereas the fields and dispatch pointer of A have been merged with those of D.

The pointer to an object encodes which of the several possible supertype interfaces the object is being viewed through. For example, a D object can be viewed as an A, B, C, or D. A D object viewed as a D is represented by a pointer to the beginning of the object, whereas a D object viewed as a C is represented by a pointer into the middle of the object, to the embedded C object within the D.

When the object is viewed as a member of one of its superclasses ("casting up"), the object pointer must be offset so that it points to the appropriate embedded object. If the superclass being cast to is the first or only superclass (class A in the figure), then the offset is zero, and can be omitted. I refer to this class as the *primary superclass*. The proper offset for converting to any of the superclasses is a constant and can be determined statically. Correct representations of the D object as B and C objects can be produced simply by adding fixed constants to the D object pointer.

For example, if `x` is a variable of type D and `y` has type B, then the Theta assignment `x := y` compiles to an assembly-language statement equivalent to

```
x := address(y) + (fields(A) + fields(D) + 1)
```

In order to generate this statement, the compiler must know the size of the fields in the classes A and D. Because the calling code uses a variable of type D, information about the structure of D and its superclass A is available — either at compile time or at link time, depending on the linkage model. If the information is available at compile time (as in C++), the fields of the superclass must be publically declared. If the resolution of the offset is delayed till link time (as in Modula-3), it becomes difficult to eliminate the add in the common case where the offset is zero.

Now, consider what happens when a method is invoked on an object. The actual class of the object may be any subclass of the known class of the expression. In addition, the method being invoked may actually be inherited from a class higher in the class hierarchy than the actual class of the object. Regardless, this method code needs to be handed a proper object of the class from which the code was inherited.

For example, consider invoking a method on an object of apparent type B. Even given just the limited class hierarchy depicted in Figure 2-5, the actual receiver object may be of class B or class D; and, even if the object is a D, the code invoked may expect an object of class B or D. Since the compiler cannot determine statically from the invocation code whether the method expects a B or a D object, the proper offset to apply to the object also cannot be determined statically.

In fact, the proper offset may be different for different methods, even on the same object. For example, the dispatch table for D may contain methods that have been inherited from any of the superclasses A, B, or C. The offset will be determined by the superclass from which the method was inherited.

Therefore, the proper pointer offsets for each method are placed in the dispatch table, and added to the object pointer as part of the dispatching operation. Figure 2-6 shows the resulting dispatch table layout. In this figure, the class `box` has overridden method `area` from class `rect`, but inherited `x` and `y` unchanged from `point`. Because there is a four-word offset between the `rect` and `box` representations of the object,

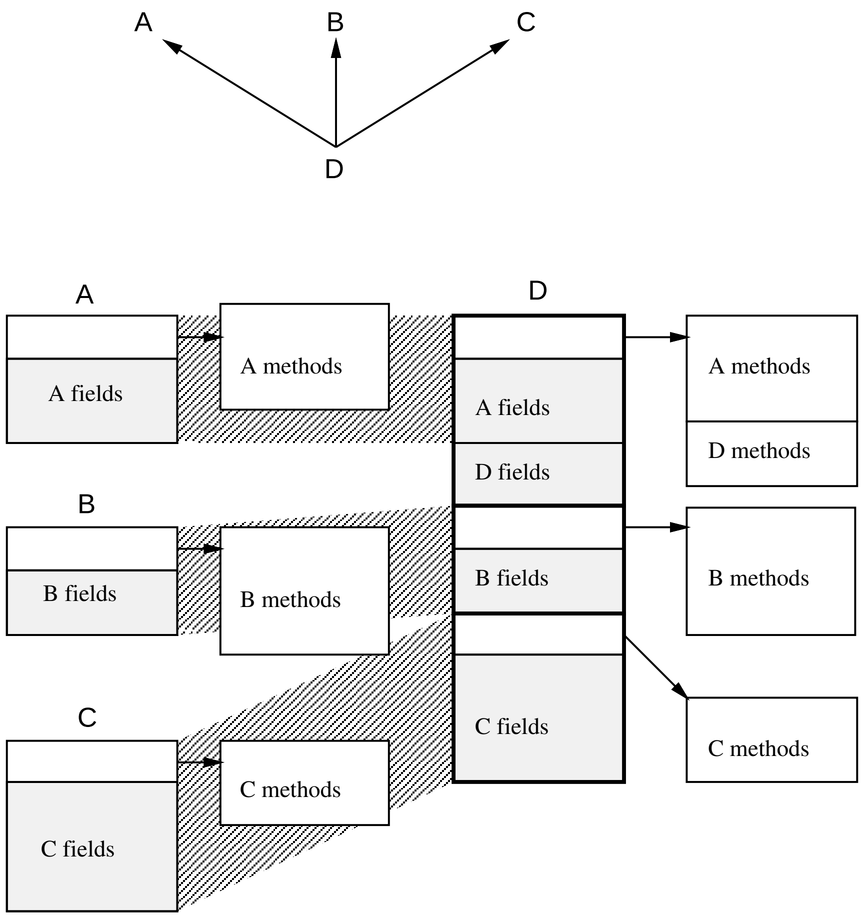

Figure 2-5: Multiple Inheritance with Embedded Objects

this offset is entered into the dispatch table for `rect` that is contained in the `box` object.

As before, I use the notation `rect.area` to denote `rect`'s implementation of `area`, and `box.area` to denote `box`'s implementation. A pointer to `box.area` shows up twice in the dispatch tables of this figure because `area` is a method of both `box` and `rect`. Each of these classes has its own dispatch table within the layout of `box`, so `box.area` appears in each of them, in the slot corresponding to the `area` method.

The dotted rectangles in the figure indicate what would happen if `box` inherited the method `rect.area` directly. The code pointers for `area` in the `box` dispatch tables are changed to point to `rect.area`. The corresponding offsets are different as well. The offset for the `box/point` section of the object is +4, and the offset for the `rect` section is 0. These offsets guarantee that the `rect.area` method always receives an object compatible with the `rect` layout.

2.4.2 The Dispatch Sequence

Loading and adding the non-constant offset from the dispatch table adds another two instructions to the dispatching process, so dispatching becomes a five-instruction sequence. Since single supertypes are more common than multiple supertypes, most of the dispatch table offsets are zero — but this cannot be known statically. Also, because each entry in the dispatch table must have its own associated offset, dispatch tables are twice as large.

On the MIPS R3000, the dispatch sequence is as shown below. I assume that a pointer to the object is located in register `t0`, and that method 3 is being called. As in the previous code example, the `t`n are temporary registers, and `a0` represents the first argument register. For simplicity, all offsets are in words rather than bytes. Note that the method index is doubled to account for the two columns in the dispatch table. In the actual machine code, some reordering of the instructions would be necessary.

```
load t1, 0(r0)      % load the dispatch table address
load t2, 6(t1)      % load the pointer to the code (6 = 2*3)
load t3, 7(t1)      % load the pointer offset
add a0, r0, t3      % offset the object pointer
jalr t2, ra         % jump to subroutine
```

Just as in the simple dispatch table model, objects are kept smaller by merging the dispatch tables of a class and its primary superclass. Otherwise, the size of an

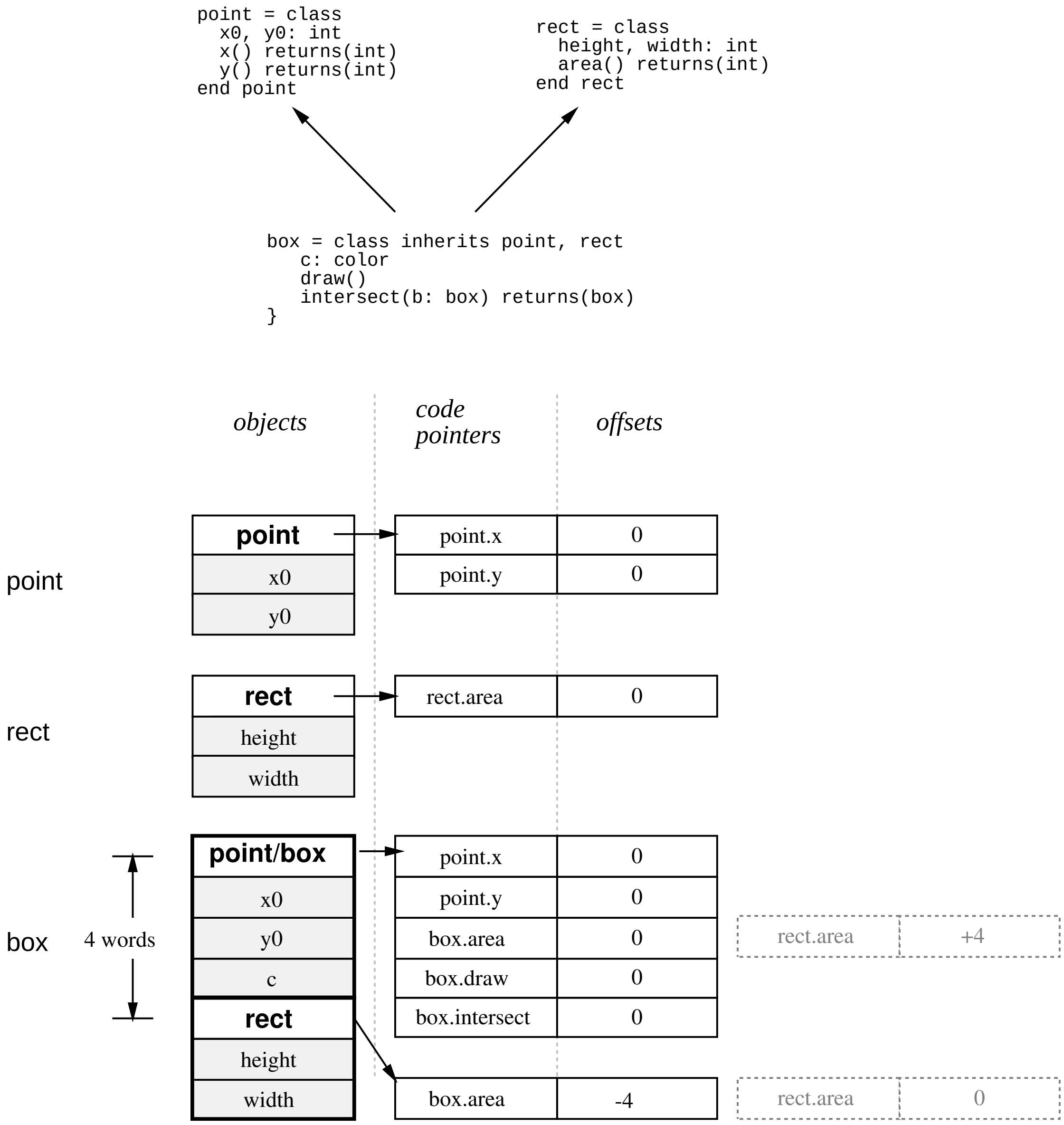

Figure 2-6: Pointer offsets in dispatch tables

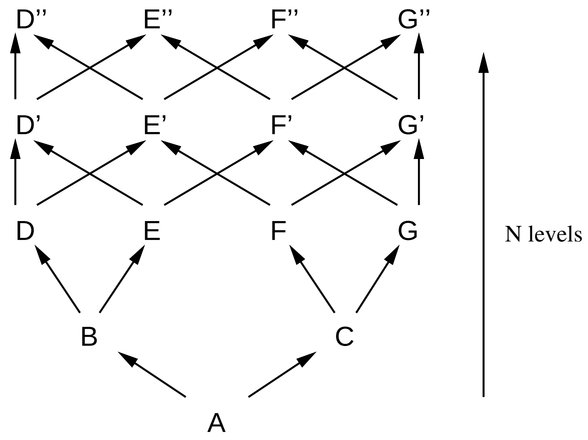

Figure 2-7: A pathological inheritance hierarchy

object would grow by one word for each level of the type hierarchy, even with only single supertypes.

Some concern has been voiced about the impact of embedded objects in a garbage-collected language, since finding the outermost object (with which the garbage collector is concerned) becomes more difficult [4]. However, this difficulty can be easily resolved by including the offset to the outermost object directly in the dispatch table. As shown in the following instruction sequence, the cost of finding the outermost object will then be 3 RISC instructions — faster than a method dispatch because no jump is required[1]:

```
load t1, 0(r0)      % load the dispatch table address
load t2, 1(t1)      % load the offset to the start of the object
add r0, t2, r0      % apply the offset to the object pointer in r0
```

2.4.3 Object Space Overhead

One problem with the C++ approach is that the use of multiple inheritance causes the total amount of dispatch data *per object* to grow. With a complex inheritance hierarchy n levels deep, the number of dispatch words per object can be exponential in the number of superclasses. An example of such a hierarchy is shown in Figure 2-7. Fortunately, such hierarchies seem to be rare.

[1]The real reason that garbage collection is difficult in C++ is that C++ supports C's & operator, allowing *arbitrary* pointers into objects.

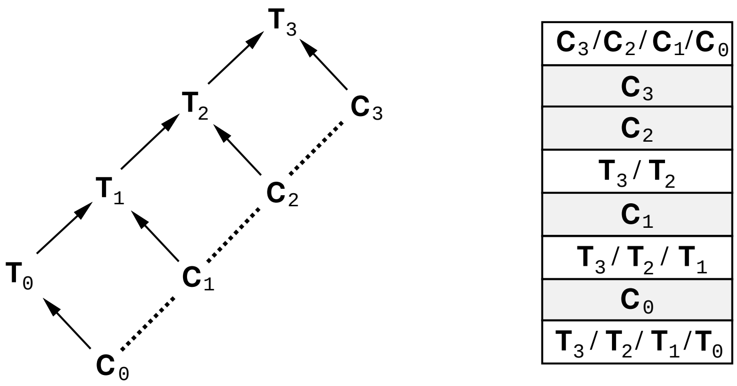

Figure 2-8: A class ladder and corresponding C++ object layout

However, a non-pathological but still problematic inheritance hierarchy, depicted in Figure 2-8, has the effect of adding one new dispatch word per level of type hierarchy. In this hierarchy, the T_i represent type interfaces or classes for which no implementation or fields have been specified. These types are related only by a subtype relationship. On the other hand, the classes C_i represent implementations of the corresponding T_i. These implementations of the more abstract types are actually sharing code through inheritance, but the code sharing is not visible to a client who only accesses objects through the T_i interfaces.

I expect this programming style to be very useful in a persistent programming environment. Because the fields of the implementations do not appear in the T_i, it provides a great deal of flexibility for reimplementing types. In a more usual C++ class hierarchy, the fields appear explicitly in the class interfaces, and new implementations cannot be added without changing the representations of all subtypes.

Unfortunately, this class hierarchy is not well-supported by the standard C++ object layout. As the object layout in the figure suggests, additional dispatch table pointers appear for each level of the inheritance hierarchy.

2.5 Bidirectional Object Layout

The bidirectional object layout I have designed for Theta addresses many of the problems outlined in the previous sections. This object layout can be viewed as a variant of the C++ approach that optimizes the case where code and fields are inherited from a single superclass. However, a type may still have multiple supertypes, preserving the interface flexibility of a multiple-inheritance system.

The object layout has three advantages over C++: first, the object layout is more compact for some important kinds of type hierarchies (and never less compact). Second, some method dispatches are faster than in the C++ layout. Finally, the object layout is better suited to persistent storage of object data.

2.5.1 An Intuitive Description

As described earlier, a C++ class with multiple superclasses has an object layout in which the primary superclass is embedded at the beginning of the object. The headers and fields of the non-primary superclasses are appended after the fields of the class. When an object is cast up to a non-primary superclass or used as a method receiver, an offset must be added to the object pointer in order to view it as one of the non-primary superclasses. When casting up, this offset is known statically by the compiler, but it must be obtained dynamically in the case of a method dispatch.

If a C++ program uses only single inheritance, all of the superclasses will be primary superclasses. All casts will be identity functions, and all offsets found in dispatch tables will be zero. Unfortunately, the dispatch sequences will still be a full 5 instructions, of which two instructions have no effect, because they increment the object pointer by zero.

Consider what happens to a C++ object if only the primary superclass is allowed to have fields (Figure 2-9). In this case, the fields of the object extend predictably at the beginning of the object. The headers of all the non-primary superclasses pile up at the end of the object, and might as well be moved to the beginning of the object, before the header of the primary superclass. As with the C++ object layout, multiple pointers may represent the same object, depending on the known type or class of the object.

The figure demonstrates the layout of Theta objects in an intuitive way. As shown, an object of class D is rearranged so that the dispatch pointers for its non-primary superclasses B and C are placed *before* the rest of the object. The effect is that the fields and dispatch pointers of the object are separated. As more levels of type hierarchy are added, the dispatch pointers grow upward and the fields grow downward.

This rearrangement of the object yields benefits in compactness of the object layout, method dispatch speed, and persistent storage. The dispatch tables can be

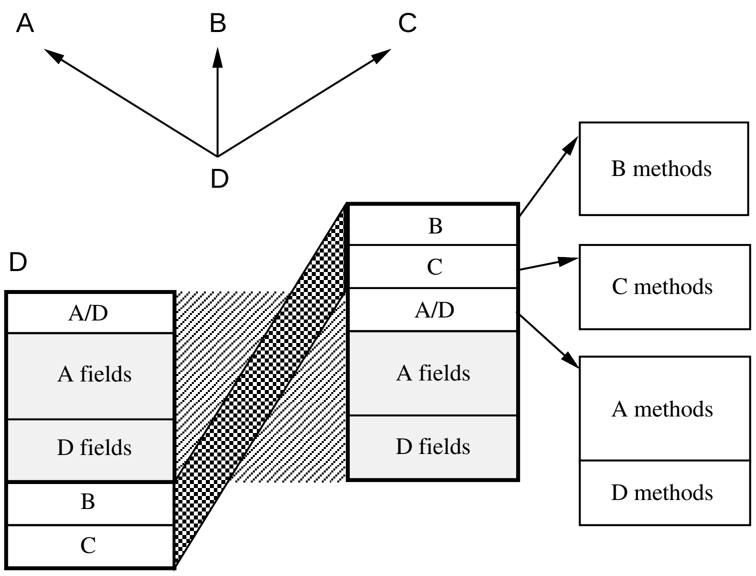

Figure 2-9: A C++ object with empty non-primary superclasses

merged in cases where the C++ layout required separate tables. Dispatches from the bottommost object pointer always have an offset of zero, so pointer offsetting can be avoided in this case. And because the data fields of the object are adjacent to one another, the persistent form of the object can be the same as the in-memory form without wasting any space.

This object layout makes the assumption that the non-primary superclasses have no fields. Before looking at the details of how the object layout works, I justify this assumption in terms of the uses of inheritance in object-oriented programming.

2.5.2 Inheritance

Multiple inheritance produces performance problems in accessing both fields and methods of an object. In a simple single-inheritance system (Section 2.3), both fields and methods can be accessed quickly because, for a particular class, they can be assigned unique small-integer offsets that hold for all subclasses of that class.

In a multiple-inheritance system, neither fields nor methods can be assigned globally unique offsets, so an additional level of indirection is required in each. In C++ implementations, fields are handled by an additional indirection when casting [2]; meth-

[2] I have omitted discussion of the indirection in field accesses, which C++ introduces when "virtual" superclasses are used, because it is not directly relevant to my object layout.

ods, by having multiple dispatch tables and embedded objects.

However, multiple inheritance is a more powerful mechanism than what most programs need. Usually, all that is needed is the ability for a type to have *multiple supertypes*. Theta supports multiple supertypes, but only single inheritance, separating the notions of subtyping and inheritance. This separation leads naturally to the object layout I propose.

One of the key insights behind the bidirectional object layout is that more efficient object layouts become possible when the system provides code-sharing inheritance for only one superclass.

In Theta, a class implements a single type; this type functions as an empty non-primary superclass in the description of Section 2.5.1. A class may also inherit from a single superclass — the primary superclass of the previous example.

Since types in Theta correspond to empty superclasses, a real object cannot be implemented by a type. A given object is always an instance of a particular class. A class has a supertype DAG that may contain branches where there are multiple supertypes. It also has a chain of superclasses, with no branches.

An example of a complex class and type hierarchy is shown in Figure 2-10. Given an object of class C, the declared type of a program variable referring to the object may be any of the types or classes depicted in the figure.

2.5.3 The Object Layout

In contrast to the rather complex C++ object layouts seen earlier, with intermixed fields and dispatch pointers, the layout of a Theta object is quite simple when expressed in terms of its type and class hierarchy. An object of class C has three parts (Figure 2-11):

- A type header. Pointers into the type header represent the object when it is declared as one of the types T_i. The type header includes dispatch tables not only for the supertypes of C, but also for the supertypes of each of the superclasses of C.

- A class header. This header consists of just a single word, pointing to the dispatch table for C. This dispatch table contains pointers to code for all the methods supported by C, including methods from C's type and also from its superclasses.

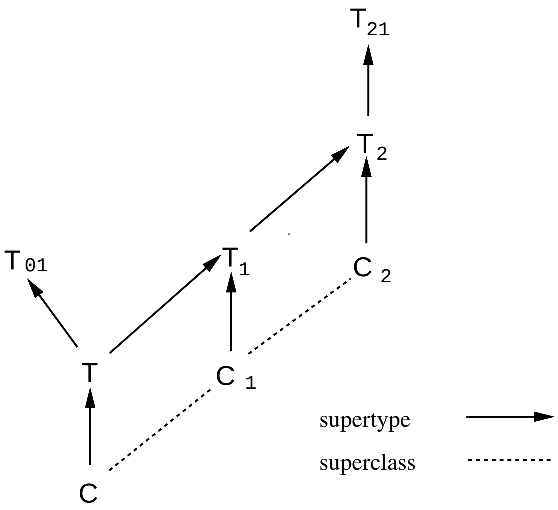

Figure 2-10: Supertypes and superclasses of a class

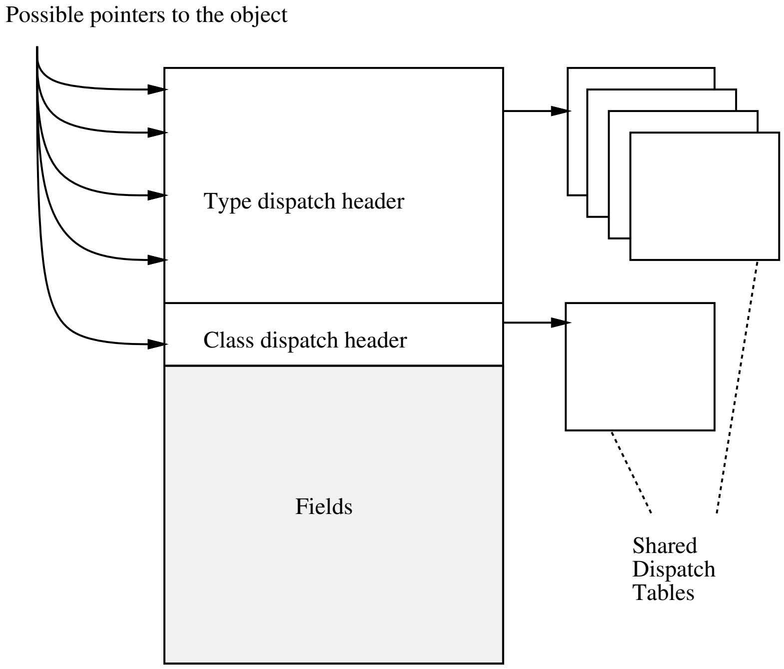

Figure 2-11: Memory layout of an object

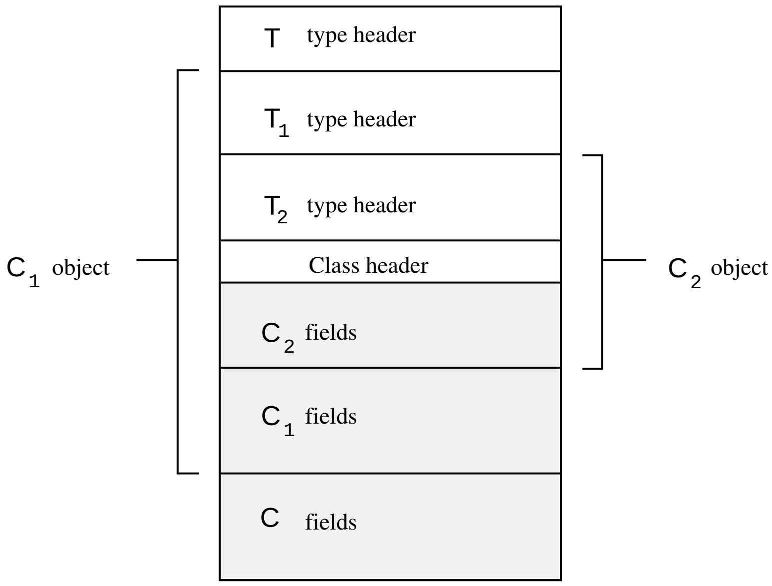

Figure 2-12: An object of class C

- The object's fields. The fields start with the fields for the topmost superclass, and proceed down the inheritance chain till the fields for C are reached.

This object structure can be viewed as being a simple single-inheritance object (the class header and fields) grafted onto a structure for supporting multiple supertypes (the type header). Unlike the C++ object layout, the multiple-supertype structure does not contain embedded fields. The class header is at the center of the object, and the successive layers of class information, both type headers and object fields, grow outward from this word. This structure is depicted in Figure 2-12, for an object of the class C shown in Figure 2-10. The innermost core of the object is in the format of the object's topmost superclass, C_2.

Consider what happens when a method is invoked on an object of a known class. In Theta, this can occur only within the implementation of C or one of its superclasses. The declared type of the object expression is either the class C or one of the superclasses C_i. Within the implementation of the method, the code will expect to see an object pointer to the same place: the class header. Therefore, the object pointer does not need to be offset when the object is of a known class. In this case, method dispatch takes three instructions, exactly as it does in the original single-inheritance C++. On a MIPS, this instruction sequence is

1. load the dispatch table address

2. load the appropriate code address from the dispatch table
3. jump through the register containing the code address

Note also that when the class of the object is known, the fields of the object are found at predictable constant offsets from the object pointer, since all embedded class objects are found at the same place in this object structure. Thus all field accesses require only a load or store instruction.

When the class of the object is not known, dispatch proceeds much as in C++ with multiple inheritance — a five-instruction sequence that offsets the object pointer by a dynamic amount before jumping to the method code.

1. load the dispatch table address
2. load the appropriate code address from the dispatch table
3. load the object pointer offset from the dispatch table
4. add the offset to the object pointer
5. jump to the register containing the code address

Unlike in C++, the object pointer is offset by the same amount for all methods in a particular dispatch table, because all methods expect to receive a pointer to the same point in the object – the class header. In C++, the need for offsets individual to each method arose from the fact that different methods required different pointers to the receiver object.

Casting an object up to a supertype or superclass is roughly as costly as in C++, since the offset between any two representations (type or class) is a statically computable constant. If the two representations are both classes, the offset will be zero, so no work needs to be done. [3]

2.5.4 The Type Header

Any type T has a dispatch header format whose size and structure depends on T's location in the type hierarchy. If the type has no branches above it — that is, T and all its supertypes have only a single supertype — the size of the dispatch header will be a single word pointing to the type's dispatch table.

[3] If multiple inheritance is used in C++, casting up is generally more expensive.

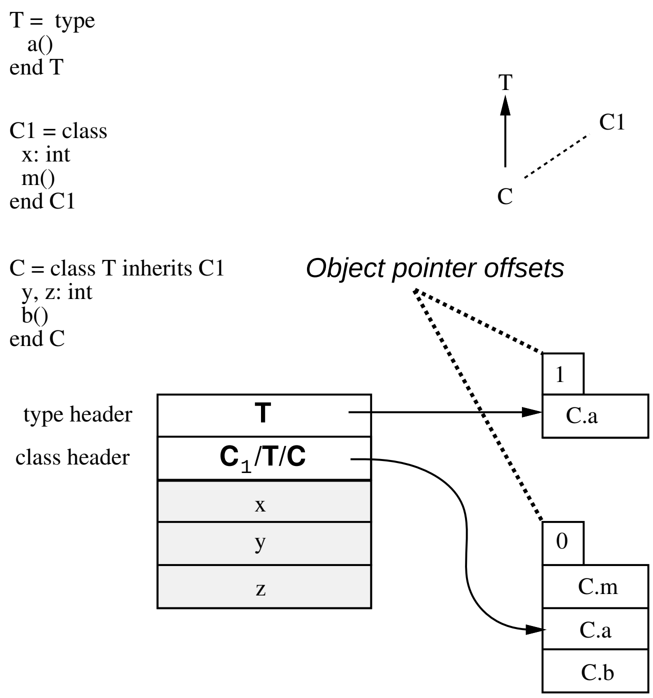

Figure 2-13: Pointer offsets in the dispatch table of an object of class C

The straightforward way to lay out a type header for type T is to place in it the dispatch headers for each of its supertypes T_i (Figure 2-14). This layout policy is similar to the policy for C++, except that no fields are intermixed with the dispatch pointers. The final entry in the dispatch header, which is also the final entry for one of T's supertypes, is *merged* with T's final entry. Thus, the size of the dispatch header for T is just the sum of the sizes of the dispatch headers for the T_i. If T has no supertypes, its dispatch header is just one word, pointing to a dispatch table for T.

Because each type has a unique and completely-determined dispatch header layout, casting an object to a supertype is either the addition of a constant, or is not required. An object pointer of type T points to the bottom of a type header for T, and headers for each of the supertypes of T are located at fixed negative offsets from that point.

2.5.5 Merging Type Headers

In general, the dispatch tables for any two types T_1 and T_2 can be merged if their common method indices always refer to the same method. As we just saw, merging

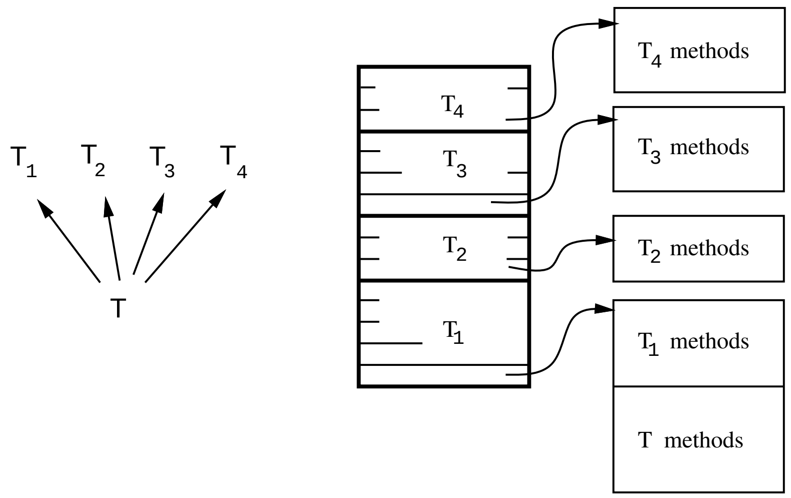

Figure 2-14: Recursively constructing type headers

becomes particularly simple in the case where T_1 is a supertype of T_2, and the method indices are assigned sequentially.

Ideally, method indices should be densely assigned, in order to keep dispatch tables small. However, sequential allocation of method indices creates conflicts in a system with multiple inheritance. Relaxing the requirement of sequential allocation, while still keeping indices reasonably dense, may allow dispatch tables to be merged more often.

The limit of merging is a system in which *every* two types can be merged. This approach requires assigning method indices based on a global analysis of the type hierarchy [11]. Unfortunately, the problem of allocating method indices for all classes while minimizing the wasted dispatch table space is NP-hard [21]. Heuristics for index assignment have been developed that reduce the wasted space to about 6%, using *two-directional record layout* [20].

The wasted space in the object layout is less important than the dependency of the method assignments on the entire type hierarchy. This dependency may be acceptable when considered as a kind of global optimization in a small, non-extensible program. However, in a system in which new types may be added dynamically, it means that every object in use could become invalid with the addition of a single type to the system. All method index assignments potentially require recomputation, so that all code and dispatch tables could have to be reworked.

In a large, distributed database, one must also question the assumption that all classes are available for the computation of method index assignments. Yet any attempt to compute method index assignment without complete information would force the FE to interrupt computation and to perform large amounts of work if the index assignments were then determined to be unworkable.

On the other hand, it seems likely that some simple heuristics for method index assignment (making dispatch tables somewhat sparse) could yield greater ability to be merged, and thus smaller objects.

For example, one such heuristic for easier merging is to populate only every nth slot in the dispatch table, starting with a slot whose index is randomly chosen in $0 \ldots n-1$. In a system where multiple supertypes are rare, the adjacent supertypes in Figure 2-14 can be merged roughly $(n-1)/n$ of the time. The effect of such a heuristic would be to reduce the average object size at the cost of increasing the total space consumed by dispatch tables. Also, having empty slots in the dispatch table decreases the effectiveness of the data cache, as a cache line contains fewer useful dispatch table entries.

2.5.6 Merging the Class Header

Even for simple, single-inheritance class structures, the Theta dispatch header is larger than one word — as currently described. The reason is that the class header plus the type header is least two words. This two-word overhead contrasts unfavorably in size with the always-one-word dispatch header of the single-inheritance layout. In addition, there is a performance loss in the casting operation, since there is an offset of 1 between the type and class representations.

In some cases, however, the class header can be merged with the last dispatch table in the type header. For example, if a class has no superclass, but a single supertype, the methods of the class can be assigned indices after the supertype's methods. This trick allows the dispatch tables to be merged (Figure 2-15). If the class inherits from a superclass, the dispatch tables cannot be merged, in general.

A common structure we expect to see in Theta is parallel ladders of classes and types (Figure 2-16). As discussed in Section 2.4.3, the usual C++ object layout introduces an amount of object overhead that is linear in the depth of the type hierarchy.

In the bidirectional object layout, the dispatch header for an object from the class ladder will have at most two words in it: one for the type header and one for the

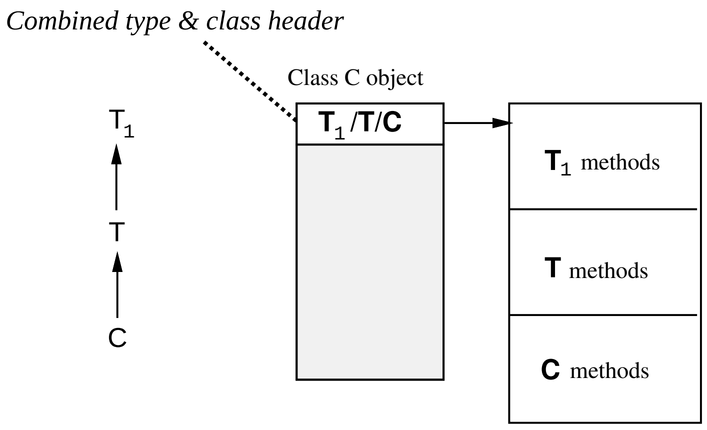

Figure 2-15: Result of merging type and class headers

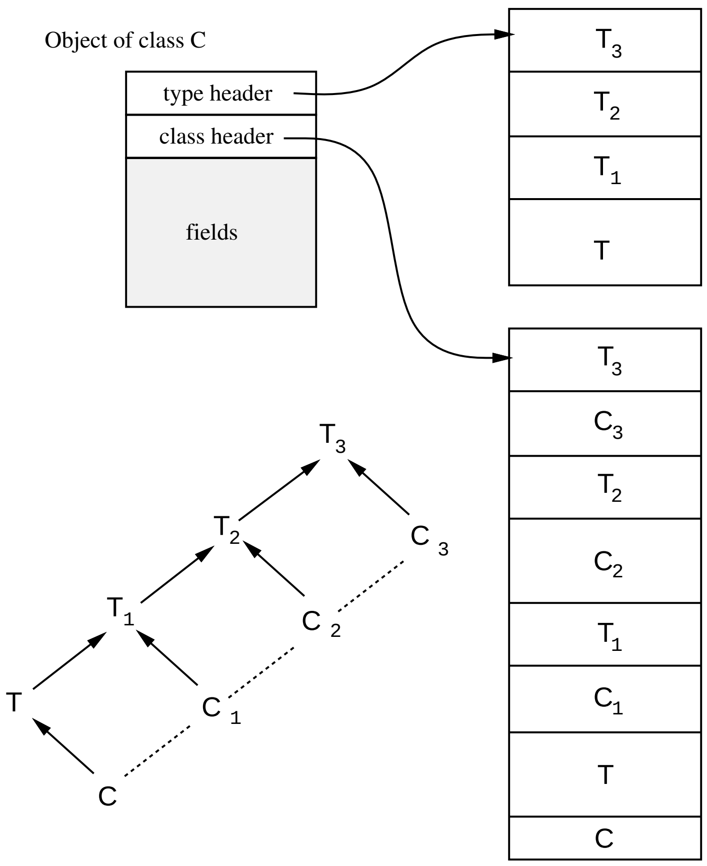

Figure 2-16: A class ladder and corresponding object layout

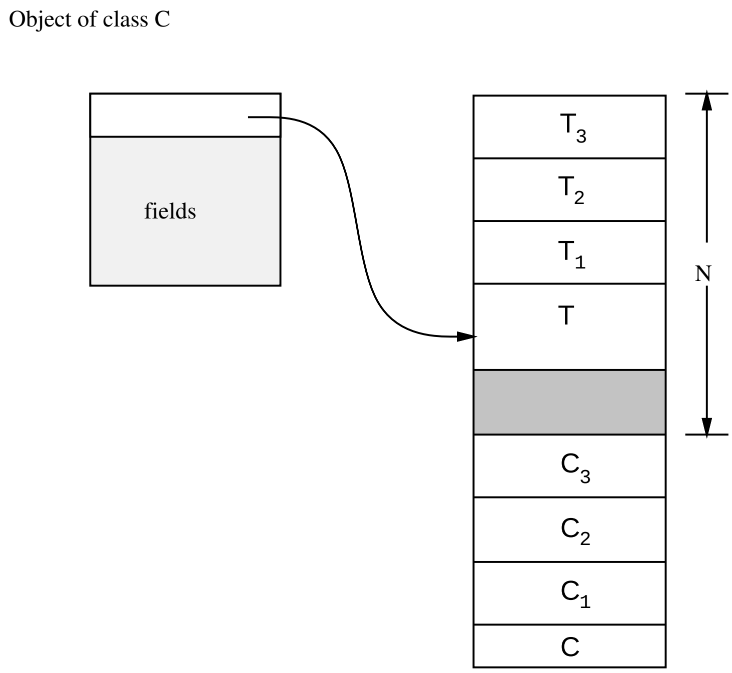

Figure 2-17: Merging the class and type header from a class ladder

class header. This object layout is depicted in Figure 2-16.

A simple heuristic for method index assignment allows these two dispatch tables to be merged, reducing the size to one word for most such classes. Method indices for class methods (methods not exported to the corresponding type) are assigned starting at some minimal index N (a value of $N = 100$ may be reasonable). This heuristic allows the type dispatch table to be merged with the class dispatch table, so long as the bottom-most type in the type ladder has at most N methods (Figure 2-17). Below this point in the type hierarchy, the two dispatch tables will have to be separated.

Obviously, N should be chosen carefully, as too large a value will waste space, and too small a value will have no effect. Since dispatch tables are shared among all objects of a class, the problem of wasted space is not critical, again assuming that the number of objects at the FE greatly exceeds the number of classes in use at the FE.

2.5.7 Avoiding Offsets

Using the optimization from the previous section to merge the type and class headers, most object dispatches will have a zero offset because most objects will have only a single dispatch word. In this case, the cost of applying the offset may be avoided by factoring the offsetting code into a separate procedure. This technique is actually used by the DEC C++ compiler `cxx`, but it has not been described elsewhere to my knowledge.

Recall that an offset must be applied to an object pointer during dispatch because the method code expects to see a view of the object consistent with the class of the method. In the dispatch sequence described earlier, two instructions are devoted to applying this offset.

The trick is to give the responsibility of applying the offset to the called code rather than to the caller. For each method, different stub procedures are created for each subclass that inherits the code. Instead of placing pointers to the real method code into dispatch tables, pointers to the stub procedures are placed there instead. A stub procedure applies an offset to the receiver-object pointer, then jumps directly to the real method code. Each stub procedure applies the offset appropriate to the class for which it has been generated.

Of course, if the offset to be applied is *zero*, then the pointer to the method code can be placed in the dispatch table, and no stub procedure is called. For these method dispatches, the cost of dispatch has been reduced to three instructions, exactly as in the simple single-inheritance dispatch.

The delay introduced by a stub procedure is small, since the stub procedure contains just two instructions:

1. offset a register by a small constant
2. perform a direct jump to the method code

Assuming that the jump does not cause any pipeline stalls (a good assumption on many architectures, because it is a direct jump), this stub procedure has only a two-instruction overhead – no more overhead than in the original dispatch sequence. Thus, this optimization is likely to improve performance.

2.5.8 Expected Performance

As long as implementations are not multiply-inherited, the bidirectional object layout dispatch technique is superior to the usual C++ layout. Costs for the various primitive object operations are shown in figure 2-18.

A cost of L represents the cost of a load from a memory location that is reasonably likely not to be in the data cache. The value of L depends on the degree of data locality in the application — basically, on the overall hit rate in the data cache. If the entire application data set (or at least the working set) fits in the data cache, the cost of L will be just a single instruction. If the application's data locality is poorer, cache misses result. On the Decstation 5000/200, using a MIPS R3000, a cache miss causes a 15 cycle delay while a new cache line is fetched [6]. At the next level, TLB misses can add another 9 cycles or so. Finally, a thrashing application will experience very large values of L as pages are swapped between primary and secondary storage.

I assume that loads from data fields or the dispatch headers of objects cost L, whereas loads from dispatch tables are cache hits, costing 1. Since a moderate-sized application manipulates perhaps tens of dispatch tables, the dispatch table data should fit into the data cache.

In addition to cache misses, loads can also generate pipeline stalls because of data dependencies. On the R3000, the result of a load is not available to the next instruction. Unfortunately, the dispatch sequence has no instruction that can be placed in the slot immediately following the load of the dispatch table pointer. If the compiler cannot move an instruction from some other computation into that position, it will be filled by a `nop` instruction, wasting a cycle. The estimated value of L must be increased by a fraction of a cycle to account for these `nop`'s.

In the row for the dispatch operation, the cost of the indirect jump instruction is written as J. An indirect jump instruction will cause a pipeline stall on some architectures. The load of the instruction following the jump must take place at the same time that the jump instruction is being decoded. Therefore, the target of the jump instruction is not available, causing a pipeline stall. The MIPS R3000 addresses this problem by always executing the instruction following the jump; this instruction is said to reside in the *branch delay slot*.

Just as with the load instruction, the delay slot can often be filled at code generation time by reordering the instruction sequence. However, the average cost of a jump of a instruction must be considered to be somewhere between one and two instructions on the MIPS. The more deeply pipelined DEC Alpha Architecture does not have delay slots, as the number of delay slots needed increases with the pipeline depth. This means that a jump through a register causes pipeline stalls on the Alpha, just as does a mis-predicted conditional branch. On this architecture, the cost of an

Operation	C++	Theta	Single-table
Dispatch	$L+3+J$	$L+3+J$ or $L+1+J$	$L+1+J$
Cast Up	L or 1 or 0	1 or 0	0
Class Field Access	L	L	L
SI Superclass Field Access	L	L	L
MI Superclass Field Access	$2L$	L	L

Figure 2-18: Operation costs in C++, Theta, and single-table systems

indirect jump is certainly larger than 1, and may be as large as 10 cycles [7].

The third column in Figure 2-18 represents a hypothetical system that assigns both methods and fields indices in such a way that all dispatch tables can be merged. This condition yields the performance of the early, single-inheritance versions of C++. It is hard to imagine how these performance numbers could be improved upon. Even copying the dispatch table into every object would would yield only a $L+J$ dispatch cost, which is little better than the $L+1+J$ shown. Except for this rather profligate use of memory, the third column represents a generally-accepted lower bound for object operation costs in a system with separate compilation.

As described earlier, the sparse method approach must examine the entire class hierarchy in order to assign method and field indices efficiently. The global nature of such a system makes it unworkable for large, distributed type hierarchies. The Theta implementation is closer than C++ to this optimum, *without* sacrificing the ability to assign method and field indices locally.

2.5.9 Persistence Considerations

The Theta object layout is well-suited to a persistent object system. All of an object's fields are packed together in the same portion of the object. For persistent storage or garbage collection — two subsystems that are concerned only with the *fields* of the object — this layout offers advantages over the C++ layout, in which dispatch pointers are found throughout the object.

The dispatch pointers of an object do not need to be stored persistently as long as the class of the object is stored, since the dispatch tables can be reconstituted from class information. If dispatch pointers are found throughout the object, there are two possible ways to store the object: leave empty regions where the pointers are normally found, or expand the objects when copying them to volatile memory in the FE. In the bidirectional object layout, the fields are contiguous and can be written to storage as a single chunk.

Similarly, in the C++ layout, a garbage collector must spend time skipping over the dispatch pointers while scanning the fields of an objects. In the bidirectional layout, there are no dispatch pointers to avoid.

The garbage collector must be able to locate the object's fields. This goal can be accomplished by placing the offset to the object's fields in each dispatch table of the object, just as is the offset to the class header. If the address of the beginning of the object is needed, an offset to it can also be placed in the dispatch table. Both of these techniques are used in the Thor implementation.

2.6 Implementation and Performance

The object layout and dispatch methodology described up to this point has been fully implemented and is being used as part of the continuing Thor implementation project. This implementation of Theta objects is written in C and runs on both Alpha-based and MIPS-based workstations from DEC. It supports dispatching on parameterized types [22, 16] as well as on the ordinary object types described here.

To test the performance model described in Section 2.5.8, I measured the overhead incurred on the MIPS R3000 when using each of the standard method call mechanisms discussed here.

The experiment was performed on a Decstation 5000/200, with a 25 MHz R3000. I measured the amount of time required to perform the dispatch mechanism used in the Theta implementation, using the optimization described in Section 2.5.7. For comparison, I also measured the performance of the dispatch mechanism used in the C++ 2.0 implementation, and the cost of two non-dispatching call mechanisms: a direct procedure call to the method, and inlining the method in the calling code.

The most interesting parameter to vary while studying the performance of these mechanisms was the number of objects in the working set. Increasing the number of objects whose data was loaded into the data cache might cause the full dispatch mechanisms to become disproportionately slower than the non-dispatch mechanisms. This effect is plausible because the full dispatch mechanisms perform an extra load on each object to obtain the dispatch table pointer. As more objects contend for the data cache, the cost of this extra load could increase. The Decstation 5000/200 is equipped with a 64K direct-mapped cache, with a line size of 4 words, and a cache

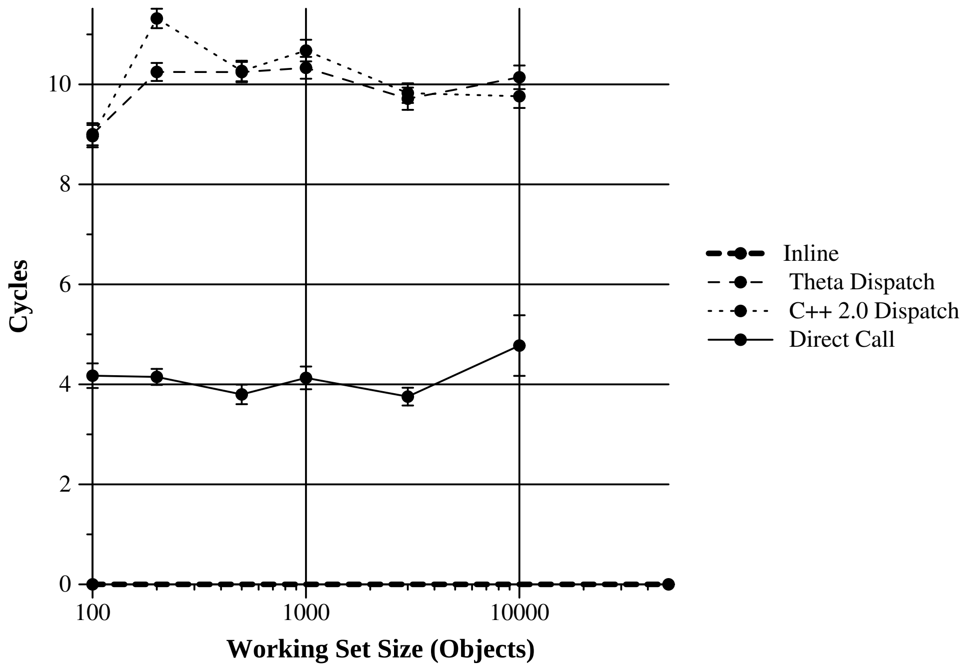

Figure 2-19: Call mechanism overhead on the MIPS R3000

miss penalty is 15 cycles [6].

The data for the experiment are shown in Figure 2-19. This graph compares four different procedure-call mechanisms, of which two are full dispatches. The lower two curves represent the overhead of a procedure call, and of inlining the procedure body. The procedure body was designed so that inlining it yielded no performance benefit other than elimination of the procedure call itself. The cost of an "inlined call" is defined to be zero, so the results for the other call mechanisms include the cost of a return instruction.

The vertical axis represents the average cost in cycles of the call mechanisms, and the horizontal axis indicates the number of objects, over which the method calls were evenly distributed.

The measured results do not show a significant difference between the two full dispatch sequences (Theta and C++ 2.0). However, the measurement technique tends to inflate the cost of the Theta dispatch mechanism because it contains three `nop` instructions in the measurement code, as compared to only one `nop` instruction in the C++ 2.0 sequence. These `nop` instructions correspond to load and branch delay slots, as discussed in Section 2.5.8.

In the measurement program, only the dispatch sequence is being measured, and so the compiler has no other useful instructions to place in these delay slots. However, in real programs, the compiler usually will be able to fill one or more of the delay slots, replacing the `nop` instructions in the benchmark code. If these `nop`'s can be replaced by useful instructions, then the Theta dispatch sequence is overestimated by three instructions, but the C++ 2.0 sequence is only overestimated by one instruction. Therefore, the Theta sequence can be expected to perform better than the C++ sequence in real use.

As can be seen from the figure, the overhead for a dispatching operation does not change significantly as the number of objects increases, despite considerable data cache contention as the number of objects exceeds 10000. The only cache effect that can be seen occurs when the number of objects in the working set is 100. With a very small working set, the performance of the full dispatch mechanisms is somewhat better. However, a working set size of 100 is unrealistically small.

Assuming that the delay slots in the full dispatch sequences can be filled with useful instructions, the Theta dispatch sequence takes about 7 cycles with a reasonable working set size. The C++ 2.0 dispatch sequence takes about 9 cycles. Note that these cycle times include the cost of returning from the called method, which the expected performance values of Figure 2-18 did not.

This lack of increased overhead can be partially attributed to the structure of the method being invoked. This method was fairly typical: it loaded a field from the object and performed a few non-memory instructions using that datum. The field access had the effect that a cache line containing the field was needed regardless of the dispatch mechanism. Since the pointer to the dispatch table was likely to be in the same cache line as the object field, the additional cost of loading the dispatch table was only a single cycle. As long as object methods access a field of the object, or call another method on the object, the overhead for using full dispatch mechanisms should not depend on the application's data locality.

On the other hand, the average cost of accessing an object increases sharply for *all* the call mechanisms as the total number of objects in the working set increases. As the working set size increases, the primary cache is unable to satisfy memory requests, and access costs are dominated by the lower levels of the memory hierarchy. Clearly, if data locality is sufficiently poor, the slowdown caused by cache misses is such that dispatch-mechanism overhead is no longer significant.

Chapter 3

Avoiding Dispatch

In this chapter, I show how method dispatch often can be avoided entirely. The cost of dispatch can be eliminated for a method invocation if the runtime linker can determine that there is only one possible implementation of the method being invoked. I explain how the Theta runtime system makes this optimization possible.

3.1 Related Work

The technique of *customization* was pioneered in the language SELF [26, 14, 25] to avoid the cost of dispatch entirely in many situations.

The idea of customization is to generate multiple *versions* of the same piece of code, depending on the environment in which the code is needed. The compiler makes customized versions of the code that take advantage of facts known in the code's caller. This contextual information allows optimization of that code version at the expense of possible code duplication — a space/time tradeoff.

3.1.1 Customization in SELF

The implementation of SELF uses customization in a way that is applicable to Theta, though in fact I use a different technique. In SELF, inherited methods are customized for each of the inheriting classes — essentially recompiling the inherited code for each inheriting environment [4, 5]. Within the customized version of each method, the implementation of the receiver object is known — it must be the class for which the method is being customized. Therefore, for any method call on the receiver, the actual code implementing its methods can be determined at compile time. Any dispatches on this object from within one of its methods can be turned directly into procedure calls, or inlined.

This technique provides performance improvements, but its applicability is limited. SELF only optimizes method invocations that occur within methods of the same class as the method being invoked — only a fraction of all the method invocations in our system. In SELF, even optimizing a small fraction of the method invocations is useful, because full dispatches are very slow. Also, since SELF treats fields and methods similarly, this customization is needed in order to obtain fast field accesses. In a system like Theta, where field accesses are not treated as method calls, and full dispatches are already fairly fast, customizing only a few of the method invocations is unlikely to yield significant performance gains.

This customization technique can also result in a code explosion, because inherited methods are implicitly recompiled for each distinct class that inherits them. This increase in code size can reduce VM performance and instruction cache performance. To reduce the effect of code duplication, SELF also uses *dynamic translation*, originated in the Deutch/Schiffman implementation of Smalltalk-80 [10]. Dynamic translation means that code is stored in a compact intermediate form and, when needed, translated on the fly into an efficiently executable form such as machine code. In SELF, customized code generation is performed on methods only when those methods are actually called using a new class. The customized code is stored in a *customized code cache* that limits the total amount of generated code. Of course, dynamic translation can also impose significant latency as code is generated.

SELF has another major optimization separate from customization for providing good numerical computation performance in a dynamically-typed system. Since Theta is statically-typed, this optimization is much less important.

Because of the limited performance gains expected and the code duplication of the SELF customization approach, it does not seem directly applicable to Theta.

3.1.2 Customization in C++

Douglas Lea has proposed that customization can be used as an extension to C++, a statically-typed and separately-compiled language [15], by adding static user annotations to C++ class definitions. While his proposal is a form of customization intended to avoid method dispatches, it is quite different from the SELF customization strategy. Unlike SELF, his proposal has not been implemented, so performance results are not available.

The use of customization in Lea's proposal is somewhat different from that in SELF. Rather than generating customized versions of inherited methods for each inheriting class, he suggests that procedure code be customized at each location where it is called, based on the types of the arguments. A special annotation, `template`, is used to indicate which procedures should be customized. Each call site generates its own version of that procedure, optimizing the procedure body according to the actual argument types passed in.

Knowing the types of the arguments to a procedure is only useful if one knows the exact implementation of those arguments. Lea proposes a type annotation, `exact`, to serve this purpose and allow customization to proceed. A variable of an `exact` class must refer to an object of exactly that class, and not of any subclass. Knowledge about exact types can be propagated using dataflow techniques similar to those in the Illinois Concert compiler [19].

The source of knowledge about exact types is objects produced by constructors, where the precise implementation returned is known. However, this knowledge is lost as soon as control passes a method-dispatch boundary. Knowledge about exact types can be preserved only by customizing called procedures for different call sites.

Lea proposes that when procedures are customized, information about the actual implementations of the return values should be used in the calling procedure. This technique allows some propagation of exact type information. The limitation of this technique in a separately-compiled language is that this information can only propagate across a procedure that has been marked for customization. In order to secure significant benefits from customization, virtually every procedure must be marked as a `template` procedure, duplicating its code at every call site. Non-trivial use of method dispatch will prevent return-value information from propagating upward, and will tend to eliminate customization.

Lea's customization proposal is intended to recover the efficiency of directly-called or inlined procedures in a system with subtyping. In his system, programmers are forced to limit the flexibility of their code in order to obtain performance benefits. The use of explicit annotations to control performance decisions is a frequent theme in C++. As noted earlier, it is not an approach that is consistent with a large, persistent object database.

Because the performance benefits of Lea's proposal are unclear, and the approach explicitly trades extensibility for performance, his work is not suitable for use in Theta. Some of his ideas for applying customization based on argument types may ultimately be useful in Theta, though, like the SELF optimizations, they involve code duplication.

3.2 Customization in Theta

I propose using the technique of customization to Theta in a way different from Chambers' application of customization to SELF. Customization is used in a way that applies to many more of the object dispatches than in SELF. The customizations used are similar to those sketched by Lea, but are performed automatically and dynamically, allowing dispatching to be avoided without any explicit user annotation.

As I will show, the embedding of the runtime system in a persistent object system allows useful customizations that would not be possible in either of the two systems discussed above.

As in SELF, the technique of customization is used to avoid full method dispatches. In fact, Theta's static typing helps greatly in identifying locations for useful customization, with the practical effect that customization becomes a more powerful and generally-applicable technique than it was in SELF.

3.2.1 The Essentials

Subtyping and the ability to reimplement an interface are very important to the long-term extensibility of a persistent object system. But in most applications, interfaces have only a single implementation. Even in object-oriented applications like user-interface toolkits, where flexibility and extensibility are important, most method calls do not require a dispatch, as suggested by the extensive use of C++ non-virtual methods in many of these toolkits.

The observation underlying my customization technique is this: if only a single implementation of a type interface is in use, there is no need to pay the cost of dispatch. Any method call on objects of that type must result in invoking the method code of that single implementation. Therefore, a direct procedure call to the code would have the same effect.

This observation leads directly to a simple and effective customization: the method call can be converted to a direct procedure call without loss of correctness. In fact, the method code can even be inlined at the call site.

Note how this use of customization differs from SELF. In SELF, code that *implements* class C is customized so that within that code, method calls on the receiver object (of class C) are made faster. In Theta, code is customized if it *uses* any type that C implements, regardless of what that code is implementing. Calls on `self` within the implementation of C are optimized according to the same condition that applies outside the implementation of C — C must have no subclasses.

Thor is a large, distributed object repository, so it is unreasonable to expect that only one implementation exists for a given type. In addition, new implementations of types may be added to the object repository at any time. How can the cost of method dispatch be avoided in such a system?

The implementation of persistence in Thor can be used as an aid to the customization process. As described earlier, the portion of Thor that contains running objects for a particular user is the *front end*, or *FE*. At a given FE, only a fraction of the total object universe is cached and immediately accessible. The FE is able to distinguish between objects that are present and objects that are not, and can determine which implementations are actively used in the FE at any given moment. This allows customization to be performed on a type that has multiple implementations, provided only one of those implementations is accessible at the FE.

Because Thor is a distributed system, multiple FEs may be running the same code simultaneously. In fact, they may even be running that code in environments that differ in the set of available implementations. These FEs may simultaneously choose different customized versions of the code, based on their differing environments.

Because an FE can be a long-running system, and new type implementations can be added to Thor, code may become *invalid*. When code that is in use becomes invalid, the FE must automatically and gracefully change the customized code in use.

Invalid code is replaced by modifying the dispatch tables, described in Chapter 2, that are associated with each class and shared by all objects of that class. Dispatch tables contain a pointer to the code that implements each method; changing these pointers changes the set of method code that implements a class. The system uses this basic mechanism to switch between method versions with different levels of customization.

The transactional model of computation in Thor makes one aspect of invalidation fairly simple. A transaction is an arbitrary computation that can be bundled up and treated as a single atomic unit that either entirely succeeds, or else has no effect. In practical terms, this means that a transactional system can roll back the system state to the beginning of the current transaction. Although aborting transactions can be used to preserve correctness, it will exact a performance penalty.

3.2.2 An Illustration

Consider a simple heterogeneous singly-linked list type named `list`, with two methods named `first` and `rest`:

```
list = type
   first() returns(any)
   rest() returns(list)
end list
```

The simplest and most common implementation of the list type is as a pair. The class `pair` implements `list` and provides a constructor, `prepend`, for producing new lists.

```
pair = class list
    head: any, tail: list

    first() returns(any)
      return self.head
    end first

    rest() returns(list)
      return self.tail
    end rest

    maker prepend(t: list, a: any)
      init { head := h, tail := t }
    end prepend
end pair
```

However, `pair` is not the only reasonable implementation of `list`. For example, classes can be implemented that efficiently provide infinite lists of repeated values, subranges of integers, and lists of items meeting some particular property — while delaying creation of list nodes until they are actually needed.

My customization techniques allow applications that use only the pair implementation to inline calls to both the `first` and `rest` methods, turning them into single load instructions. If the FE then encounters a special infinite list object, all dependent dispatch tables will be modified, gracefully changing all calls on list methods to full method dispatches.

For example, the type `stack` is conveniently implemented in terms of lists. Below, we see the definition of the type `stack` and the class `list_stack` that implements it. In Theta, the expression `list.prepend` invokes the constructor `pair.prepend` defined above. A call to `list.prepend` is implemented as an ordinary procedure call.

```
stack = type
     push(x: any)
     pop() returns (any)
end stack

list_stack = class stack
     items: list

     push(x: any) items := list.prepend(items, x) end push

     pop() returns(any)
        l: list := items
        top: any := l.first()
        items := l.rest()
        return(top)
     end pop

     ...

end list_stack
```

Clearly, this implementation of the pop method would be much faster if the list methods `first` and `rest` were inlined, as it would become the code

```
pop() returns(any)
   l: pair := self.items         % load
   top: any := l.head            % load
   items := l.tail               % load, store
   return(top)
end pop
```

which is essentially just three load instructions, a store, and a return. This method implementation will be substantially slower with the overhead incurred by using full method dispatch (at least 12 additional instructions: two method dispatches, two return instructions, and some register saves) Therefore, the implementation of `list_stack.pop` will run about three times faster if the list methods can be inlined.

Two versions of `list_stack.pop` will exist: a generic version that performs full dispatches on list objects, and a customized version that assumes list objects are `pair`'s. Only one of these versions will be in use at any given time. The version in use will be installed in the shared `list_stack` dispatch table (and the dispatch tables of any classes that inherit `list_stack.pop`).

This example demonstrates the basic idea behind customization. However, many of the details of the system remain to be described. In Section 3.3, I look at the structures needed to maintain a database of customized code and select from it. In Section 3.4, I show how the cache of currently-used customized code is maintained and code is invalidated when necessary. Finally, in Sections 3.5 and 3.6, I discuss the performance of this approach and the state of the current implementation.

3.3 Producing Code

One piece of customized code can only call another piece of customized code directly if the conditions for customizing the called code are met. Customized callers are indirectly dependent on the same conditions required for customization of the callees. This dependency propagation has implications for the management and creation of customized code. Consider the following example.

3.3.1 Propagation

In the example of `list_stack` and `list` from Section 3.2.2, consider what happens if `list_stack` has no subclasses.

Other code — say, the `parser` class — may be customized on the uniqueness of the stack implementation — either calling `list_stack.pop` directly, or else inlining it. In either case, a choice must be made about which `list_stack.pop` code to use: the generic version or the customized version. Note that the generic version may always be used, but that the customized version can be used whenever it is valid, in order to improve performance.

If the customized version of `list_stack.pop` (which depends on the assumption that `pair` is the unique implementation of `list`) is used, then the correctness of `parser` will also depend on the assumption that `list` has only a single implementation. The dependency will have *propagated* up another calling level, even though the implementor of `parser` may not know even that `list` is used.

This propagation of customization dependencies has both its good and bad aspects. On the one hand, it means that multiple levels of inlining can take place, even across object dispatches, with consequent opportunities for performance enhancement. Inlining across abstraction boundaries improves performance without sacrificing abstraction.

However, dependency propagation also causes larger and larger amounts of code to depend on a particular assumption about the state of the front end — an assumption whose validity might change at any time.

Avoiding dependency propagation is easy — as described, one simply uses the generic version of a unique implementation, rather than the most customized version available. Using the generic version would probably be preferable when the gain from using a more customized version is small compared to the time spent in the customized version.

3.3.2 Bookkeeping

The Theta runtime system maintains information about the dynamically available types and implementations in order to allow customization.

A given class, such as `list_stack`, stores information about each of the methods it implements. These methods are stored in a generic intermediate code form in which all object methods, including methods of primitive types (e.g. `int`), are represented as full method calls. This generic intermediate code is not executable; in fact, it is architecture-independent.

A special code generation phase, the *customizer*, takes in generic intermediate code and some information about the environment in which the method is being used, and produces customized versions of the code that take advantage of types with only one available implementation. These customized versions contain executable machine code.

The customized code versions produced by the customizer are stored in the class that owns the method. Each class maintains information about each of the available customized versions of its methods (Figure 3-1). In this figure, we see that `list_stack.pop` has two customized versions, whereas `list_stack.push` has only one. However, the intermediate code for each method is stored only once.

Customized versions are stored on a per-method basis, rather than on a per-

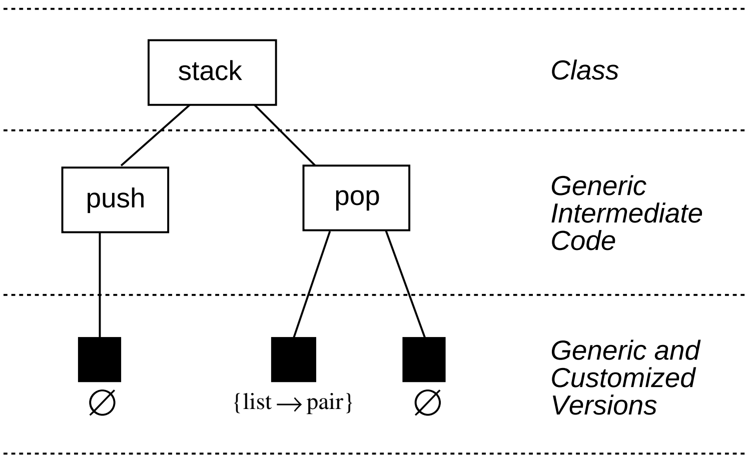

Figure 3-1: The information attached to methods

class basis. Customizing methods independently is best because the set of useful customizations depends on what types and classes those methods use. Different methods use different underlying types, so they should be customized differently. For example, customizing `list_stack.push` with the assumption `list`→`pair` would be pointless. Since `push` does not invoke any list methods (`pair.prepend` is not a method), customizing it would yield exactly the same code as the generic version.

Customized versions, like the intermediate code, are stored persistently and fetched from the object repository when needed. Unlike the intermediate code, customized versions may also be discarded without loss of correctness. Given an object at the FE, one or or more customized versions may be present at the FE at a time for each of the methods in its class, though usually only one version is present. For classes not being used at the FE, none of the class's code is present.

3.3.3 Assumptions

Each customized version keeps a set of the *assumptions* on which it is customized. An assumption is simply a condition of the runtime system or calling environment that allows some customizations to be performed. A customized version is *valid* if all the assumptions used to customized it are true in the current environment. Because a customized version lists *all* of the assumptions on which it depends, the runtime system can pick out the customized versions that are currently valid.

The customizer generates this set of requisite assumptions. Thus, customization can be described as a function that takes in a piece of generic intermediate code,

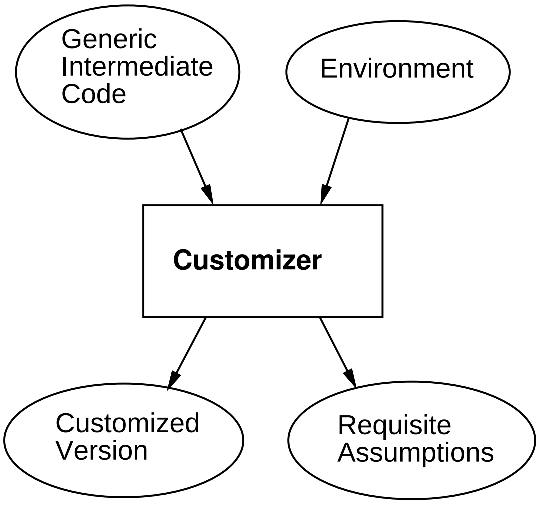

Figure 3-2: The customizer

and produces customized code to which is attached the complete set of assumptions used to produce the code. Because of the propagation of assumptions, the set of attached assumptions may include some *hidden* assumptions — assumptions derived from called code. The customizer is represented schematically in Figure 3-2.

Each method always has a customized version in which the only customizations are to eliminate dispatch for primitive types such as integers. Primitive built-in types never cause a dispatch because the customizer always recognizes them as opportunities for customization. These *generic versions* are labeled $\emptyset$ in Figure 3-1 — since a generic version is always valid, its set of requisite assumptions is empty. I use the notation $T \rightarrow C$ to denote an assumption that C is the unique implementation of T. For example, the notation `list`→`pair` denotes the assumption that `pair` is the only available implementation of `list`.

In order to determine when a method call can be optimized, types are annotated by the FE to indicate whether there is only a single implementation of that type active at the FE. Similarly, classes are annotated to indicate whether they have any subclasses. Consider the example of Section 3.2.2. When the customizer examines a call on a method of type `list`, it can easily check the annotations for `list` and discover that `pair` is its only implementation. This annotation requires only a small, fixed amount of space per type. Note that any subclass of `pair` would also be considered an implementation of `list` when maintaining annotations.

When a piece of customized code is produced, the customizer considers each method dispatch to determine whether it can be optimized into a procedure call. The annotations on the type of the receiver expression are examined to determine whether that type has a unique implementation.

When a method call found in the generic intermediate code is optimized, substituting for it a direct call on a customized procedure P (or inlining P), any assumptions attached to P are automatically propagated to the calling code.

Note that if method calls are customized into calls to the generic versions, the customized code does not acquire any hidden assumptions. The generic versions are not dependent on any assumptions, so no additional assumptions propagate to the caller. The set of output assumptions only contains assumptions that correspond to the types directly used in the method.

3.3.4 When to Customize

In the previous section, I described how customization is performed. Here, I discuss the problem of when to customize.

The system decides whether to customize when it populates a class dispatch table with method code pointers. For each method in the dispatch table, there are a number of customized versions, at least one of which is valid. Each customized version corresponds to a distinct code pointer that could be installed in the dispatch table. The system may either install one of the existing valid versions or produce a new version and install it.

The philosophy for customization in the current implementation is to customize aggressively, perhaps resulting in too much code duplication. A new customized version is generated when the customizer can determine that it would have fewer full dispatches than in any valid, existing version.

As part of the production of generic intermediate code for each method, a special *type use* compiler phase processes the intermediate code and identifies all the distinct types (and classes) that are used in that method. The intermediate code is annotated with this information for later use by the customization decision process. For example, when the compiler examines the method `list_stack.pop`, it determines that the type `list` is used in that method. Therefore, customization on `list` may be useful when `list` has only one implementation, and so `list_stack.pop` is annotated to indicate this fact.

Whenever a customization is being considered, some subset of the types used in the intermediate code have only one implementation. In the current prototype customizer, a new customized version is generated when no existing customized version takes advantage of all of the types in this subset.

Although I term this strategy “aggressive,” it may miss some opportunities for useful customization. For example, it may fail to generate a version that fully optimizes inlined code, because the type use phase analyzes only the intermediate code of the method, before any inlining. Types used in the inlined code are not necessarily used in the intermediate code, and may not be recognized as opportunities for customization.

The aggressive strategy is almost certainly overkill. Customizing most methods will produce a negligible *overall* performance benefit, simply because most time is spent in a fraction of the code. In fact, customizing some methods may reduce overall system performance, for one of the following reasons:

- Code duplication: The repository might end up storing many different versions of the same code, which will compete for space with the FE's cache for storing other objects.

- Latency: Compiling takes time, especially when optimizing extensively. The opportunity to produce new code versions occurs in the middle of computation — a particularly bad time for a lengthy interruption.

- Invalidation: The more customized a method is, the more likely that it will be invalidated, and invalidation can be expensive. In the worst case, invalidation could happen repeatedly on the same method, with a less-customized version being selected at each iteration. This scenario could conceivably occur at application startup time, when new type implementations are being rapidly added to the FE.

For these reasons, methods must be selected carefully for customization. The effect of customizing a method is to make it run faster, since dispatch sequences within it will be simplified. Therefore, the methods most worth customizing are those methods in which the most time is spent. Unfortunately, the compiler cannot determine *a priori* which methods are performance-critical.

One way to select methods worth customizing is to use profiling information. Using profiling feedback to automatically control optimization is not a new idea; the MIPS code-ordering tool `cord` is an example of such a code optimizer.

In fact, the system can automatically profile itself by sampling the program counter periodically as it performs computation. The sample program counters can then be mapped back to the method code they correspond to and used to identify the performance-critical methods that are worth customizing. Simple experiments indicate that the cost for using PC sampling with 10 ms resolution is less than 1% on a DEC Alpha running OSF 1.2, so using PC sampling at all times may be reasonable on modern architectures.

Since only a fraction of the methods will be customized, the problem of code duplication will be greatly lessened. Also, customization will not occur until a generic version has been used many times, so repeated invalidation is made much less likely.

Finally, the problem of latency can be addressed by performing customization in background. The production of new customized versions based on the profiling information can be delayed until the FE is not busy with client requests.

3.3.5 Inlining

Two options have been mentioned for optimizing method calls: direct procedure calls and inlining. From the standpoint of customization and assumption propagation, these two options are identical. Therefore, inlining in Theta creates no special new problems that are not associated with other dynamically compiling systems, such as SELF. Inlining is performed at the intermediate-code level of customized code, and can provide substantial performance improvements for short methods.

3.4 Invalidation

Thor is a dynamic system. Consider the example of Section 3.2.2. Even if `pair` was the only implementation of `list` when the dispatch tables of `list_stack` were initialized, objects of a second `list` implementation may be fetched at any time. As soon as such objects are known to exist, the dispatch table for `list_stack` becomes *invalid* and must be updated to point to the generic versions, or at least to versions of the `list_stack` methods that do not depend on `list`→`pair`.

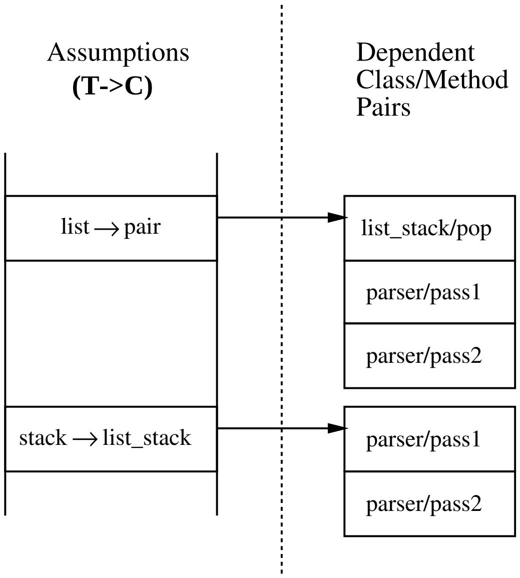

Figure 3-3: Assumption registry

3.4.1 The Assumption Registry

The dispatch tables of classes contain pointers to appropriate versions of the corresponding methods, so the correctness of these tables is dependent on the validity of the assumptions that were used in the installed method versions. When an assumption becomes invalid (because a new implementation of a type has been fetched into the FE), the system must be able to find all the dispatch tables that are dependent on a particular unique-implementation assumption.

This functionality is provided by the *assumption registry*: a mapping from assumptions about the runtime state to classes whose dispatch tables are dependent on those assumptions (Figure 3-3).

For example, the assumption registry allows rapid location of all the classes whose dispatch tables are dependent on `list`→`pair`, so these dispatch tables can be updated to point to valid versions of the methods.

When the dispatch tables of a class are filled in, a class/method pair is entered in the assumption registry for each assumption used in any of the method versions installed in the dispatch table.

In an earlier example, the class `list_stack` implemented the type `stack` by using the type `list`. The `pop` method of `list_stack` may be dependent (as we saw earlier) on `list`→`pair`, because it uses methods of the type `list`:

```
list_stack = class stack
   ...
   pop() returns(any)
      l: list := items
      top: any := l.first()
      items := l.rest()
      return(top)
   end pop
   ...
```

We hypothesized a class parser whose implementation uses stack.pop, perhaps in the pass1 and pass2 methods. Because of assumption propagation, these methods of parser may also be dependent on the list→pair assumption, as depicted in the figure.

If the dispatch table of a class is updated, the assumption registry must also be updated to reflect the new situation. Since the class contains the set of assumptions used by each version of its methods, the class/method pair can be removed from the registry wherever it occurs and re-entered with the information corresponding to the currently installed method versions.

3.4.2 Detecting and Handling Invalidation

Even running code may become invalid. For example, a method customized to assume only one implementation may cause an object of a second implementation to be fetched. If the customized method then invokes a method on the new object using the old assumption, it will directly call the original implementation's code, handing it an object of a different implementation. Obviously this must be prevented.

It is straightforward to determine that invalidation is necessary. The Theta runtime system can detect when a new implementation is being used by objects in the FE cache. When the implementation becomes available, the runtime system scans up the type hierarchy, examining every supertype or superclass of this new implementation. Since every type is annotated to indicate whether it has a unique implementation, it is easy to determine whether some current code may need to be invalidated.

I make some assumptions about what changes are acceptable as part of invalidation. Changes that scale with the number of accessed objects are not acceptable. For example, changing the dispatch headers in every object is a bad strategy. I also assume that invalidation is infrequent.

There are three important ways that the runtime system can be using customized code when it becomes invalid.

- The invalid code can be executing. Conceivably the runtime system could patch both the current program counter and the return addresses in the program's stack. But even if the invalidated code is not the currently executing procedure, the new version of the code may have a different stack layout, and returning the flow of control into that stack frame will cause havoc. Mapping the old stack frame layout into the new layout is tricky, especially in the face of arbitrary optimizations and inlining. This technique is used in SELF, but creates significant implementation problems [3]. Using this technique in Thor would make it less portable, since rewriting the stack usually depends on details of the machine architecture.

  Another approach is to abort the current computation (erasing its effects) and restart it from scratch. Aborting the current computation is only *required* if one of the procedures on the stack is invalid, though technically an abort is always a safe maneuver. Avoiding aborts is desirable for performance reasons, since an abort means that some computation must be repeated.

  Aborting requires runtime support for transactions, so that objects modified by the partially-completed computation can be reverted to their pre-computation state. Because Thor is a persistent, distributed system, transactions are already a part of the Theta computational model.

- Class dispatch tables may contain pointers to the invalid code. This problem is easily solved by rewriting the appropriate entry of the dispatch table to point to a valid version of the code. Since dispatch tables are shared by all objects of that implementation, the cost does not grow with the number of objects.

  When a second implementation of a type T becomes available, the assumption registry is checked to see if the dispatch tables of any classes are dependent on the assumption `T`$\rightarrow$`I`, where `I` is the current unique implementation. The dispatch tables for dependent classes are rebuilt under the new assumption set, replacing affected methods with new, valid customized versions. The assumption registry is then updated appropriately.

  The new version of the code installed in a dispatch table always can be the generic version of that method — the always-valid version that is based on no assumptions, and only customizes operations on primitive types. Using this version allows computation to continue immediately in the case that the most-customized valid version is not available at the FE.

- Other code may contain direct pointers to the invalid code, or inlined versions of it. This can happen in two ways: either the other code has been customized and a method call was optimized to invoke the code directly, or else the invalid code is a procedure that is being called in the usual way, like `list.prepend` in the earlier example.

  In either case, assumption propagation will ensure that the other code will also be dependent on the same assumptions that were used to generate the invalid code, so both the caller and the callee will be invalidated. No special action is needed for this use of customized code.

When an assumption is invalidated, the runtime system needs to locate all of the invalid code currently in use. It must rewrite any dispatch tables pointing to the code, and restart the current transaction if any invalid code is on the call stack.

To handle running code that has been invalidated, the runtime system must be able to determine what code is currently executing. This is accomplished by walking up the stack and extracting the program counter values that have been pushed onto it when procedure calls were made. The actual executing code is then located using these PC values. As mentioned earlier, the machinery for mapping PC values back to the corresponding code objects is also useful for automatic profiling.

3.5 Performance

The effects of optimizing away dispatch compare well to the performance results from section 2.5.8. For example, when a dispatch is converted into a procedure call, it becomes a two-instruction sequence: one instruction to properly offset the object pointer as described in section 2.4.1, and a second instruction to jump directly to the implementing code.

In the full dispatch sequence on an object of type T, the object pointer was added to an offset that had to be loaded from the dispatch table. In the optimized dispatch sequence, this offset must still be applied to the object pointer. However, the object offset can be determined statically, because the unique implementation of T is known.

Operation	Theta	Optimum	Direct Call	Inline
Dispatch	$L+3+J$ or $L+1+J$	$L+1+J$	1 or 2	-1

Figure 3-4: Dispatch costs, revisited

The proper object offset is just the inverse of the offset applied to cast an object of this implementation to type T. This inverse offset is a peculiarly efficient implementation of casting down. In many cases, the object offset will be zero, and can be avoided altogether.

In terms of the constants defined in section 2.5.8, the cost of a procedure call dispatch is 2 (or 1 if the offset is zero). The J cost is avoided because the direct, unconditional branch allows the pipeline to process instructions in the subroutine without stalling.

Inlining has a negative cost, because no return instruction is needed. It also allows the procedure body to be optimized in its calling context, so the compiler can avoid the cost of setting up argument and return-value registers. This benefit is dependent on the number of arguments and return values. Inlining code also increases the amount of code in the system, possibly making the instruction cache less effective and counteracting the more obvious benefits in a way that is difficult to quantify.

Note that both inlining and direct calls bring the cost of dispatch below that of the "optimum" column, as shown in figure 3-4. This fact suggests that judicious use of customization might bring the average cost of method dispatches, including those dispatches that cannot be avoided, down to the point where the performance of Theta is similar to that of non-object-oriented languages.

3.6 Implementation

The implementation of customization has been prototyped in Modula-3, including a customizing compiler that aggressively converts generalized intermediate code into customized code by performing both inlining and direct-call substitution.

The current prototype implementation does not perform automatic inlining. Inlining is performed according to a user directive, and on methods of the primitive types such as integers. This user-directed inlining is still more flexible than C++-style inlining, as customization allows it to be performed even when multiple implementations of a type exist.

I have also implemented a code generator that converts customized code into working C code. Although the customizing compiler works, certain key pieces of the running system are still missing. The profiling support and the facilities for dynamically loading, managing, and relocating code objects are incomplete. However, I feel that the level of design and prototyping that I have done are enough to indicate that this customization technique is both feasible and useful.

Chapter 4

Conclusion

I have presented two independent but complementary techniques for speeding up object operations in a persistent programming language. Together, these techniques address much of the concern for performance in a persistent programming language.

In Chapter 2, I described a novel object layout that speeds up method dispatch and field accesses as compared to the usual C++ layout. The core idea is to take advantage of the separation of inheritance and subtyping. When code-sharing inheritance is allowed only with a single superclass, objects can be rearranged to make object operations more efficient. For persistent and garbage-collected systems, the object layout provides additional benefits because the fields of the object are packed together. I also showed how multiple inheritance can be implemented in this scheme without slowing down single inheritance.

I implemented this object layout as part of the Thor system, currently under development. The measured performance results for this object layout indicate that dispatch is fast and causes minimal extra load on the data cache.

In Chapter 3, I showed how customization can be applied to a statically-typed, persistent object-oriented language to avoid many method dispatches. The called code can be called via the ordinary procedure-call mechanism, or inlined directly in the calling code. The statically-typed nature of Theta makes it possible to avoid dispatches that could not be avoided in SELF. The fact that Thor is a persistent object system is exploited to eliminate dispatches based on the state of the current computation at the FE, so that rarely-used additional implementations do not vitiate dispatch avoidance.

I wrote a prototype customizing compiler for Theta intermediate code that eliminates method dispatches when possible, replacing them either with direct procedure calls or the inlined code of the method being called.

There are many ways in which this work can be extended or improved.

4.1 Merging Type Headers

One of the defects of the object layout scheme described in Chapter 2 is that the per-object dispatch information can become fairly large if the type hierarchy is complex — a problem shared by C++. This problem is exacerbated by the presence of surrogates, which have all the per-object dispatch information of the real objects, without the fields. I expect that this storage overhead will not be a problem in practice, because the most common objects tend to have simple types. As I have shown, only one or two words of dispatch information are contained in objects from type hierarchies with single supertypes.

However, it would be desirable to reduce the dispatch overhead in objects from more complex hierarchies. As suggested in Section 2.5.4, some simple techniques for assigning elements in the dispatch table can yield greater mergeability of dispatch table. These technique would reduce the per-object overhead while allowing indices to be assigned locally — in contrast to the sparse table schemes, which are unsuitable for a large system.

More investigation is needed to determine the right tradeoff between sparseness in the dispatch table, locality of index assignment, and the performance of the data cache in the presence of sparse tables.

4.2 Customization

More performance studies of the customizing compiler would be helpful in understanding some of the issues of the design. Although I have prototyped the customizer, the Theta compiler is incomplete.

Some of the issues that performance measurements would be helpful in resolving are:

- When should new customized versions of a method be generated, in order to optimize performance?

- How much assumption propagation should the customizer allow?
- When should inlining be used?

4.3 Individual-Method Customization

In the work described, customization can be performed when a type has a single implementation, represented by the notation $T \rightarrow I$. Another way to customize is to notice when a single method has a unique implementation, and substitute direct procedure calls for the method calls on that method. While all methods have the potential for being overridden by a subclass, some methods are never actually overridden. Therefore, more call sites will be customizable under this approach. This extension could provide performance equivalent to non-virtual methods in C++, though it would also increase the complexity of the system and possibly cause more invalidations.

4.4 Customizing Surrogate Checks

The customization machinery described in Chapter 3 can be used to optimize operations other than method dispatches. The bookkeeping of assumptions and dependencies, and the techniques for recovering from invalidation, apply to other optimizations as well.

To implement orthogonal persistence, Thor uses *surrogate objects* [17]. A surrogate object exists only at the FE, as a placeholder for a real, persistent object. It is able to accept method calls, like the object that it represents, but it does not contain the real object's data. Instead, invoking a method on the surrogate object causes the real object to be fetched from the object repository.

Before any operation can be performed on an object, a *surrogate check* must be performed to determine whether the object is actually a surrogate, and if so, convert it into a real object. Ultimately, references to the surrogate are patched by the garbage collector, redirecting them to the true object. Surrogates can be also be created by the FE when memory is scarce — the FE *shrinks* objects, replacing them with surrogates [17].

Surrogate checks are a performance problem unique to persistent languages. But in many contexts, an object is known not to be a surrogate. For example, the receiver object of a method is never a surrogate, nor is an object just produced by a constructor.

In order to avoid surrogate checks, the compiler can annotate the types of expressions to keep track of which expressions might refer to surrogates. This annotation looks like a simple extension to the type system used by the compiler. These surrogate annotations are propagated through the intermediate code, so as long as it is not assigned to, a variable is only checked once to see if it refers to a surrogate.

There are three major weaknesses of this approach:

1. The FE can shrink objects asynchronously, so reasoning about surrogate annotations may be difficult.

2. Surrogate annotations are lost when making a method call, unless that method is inlined. If the method was called directly, it does not know whether the arguments to the method are surrogates or not, and must assume the worst.

3. Unlike variables, object fields are difficult to usefully annotate, because any method call might result in a change to the value of that field. Therefore, they are a major source of surrogate checks.

The problem of shrinking can be addressed by preventing the FE from shrinking objects that running code might care about. For example, local variables can be prevented from turning into surrogates if the FE checks the stack and register set before turning an object into a surrogate.

To avoid the second limitation, the system can use customization on procedure argument types, allowing surrogate information to cross procedure boundaries at the cost of duplicating the called code. The idea is that different callers use different versions of the code, depending on which of their arguments are known not to be surrogates. The idea of automatically customizing on procedure argument types has been suggested as an extension to customization in SELF [4], although the idea was not seriously explored there. Considerably more work needs to be done here to understand the time/space tradeoffs and the additional runtime bookkeeping.

Avoiding the problem with object fields is more problematic. In order to know that a field is not a surrogate, the compiler must be able to determine that the field is unchanged since the last time it was checked. More investigation needs to be done to determine when this is feasible in a system like Thor.

4.5 Class Evolution

Very long-term extensibility in Theta requires a form of schema reconfiguration: objects should be able to change their implementations without changing their identities. One quasi-linguistic mechanism toward this end is class evolution — allowing changes to an existing class. Ideally, it should be possible to completely change the implementation of a class, adding or removing fields or private methods.

Class evolution can be effected by obsoleting one class, and providing a description of how to convert objects of that class into objects of the new class. Simultaneously changing the class of all objects in a large distributed system is obviously infeasible. However, this conversion can be performed lazily at the time that the FE fetches objects from the OR. If surrogates are equipped with appropriate dispatch tables, class conversion can even be delayed until the object is actually used, reducing the cost of class evolution even more.

However, class conversion will also affect the use of customization, since customized methods may use information about the old class in order to speed up code. For example, customized methods may directly call the methods of the old class, or even inline their implementation. Further investigation is needed to understand what effect the presence of class evolution has on the customizations described here.

Bibliography

[1] P.S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *OOPSLA '89 Conference Proceedings*, pages 457–467, October 1989.

[2] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Language definition. In Greg Nelson, editor, *Systems Programming in Modula-3*, chapter 2. Prentice-Hall, 1991.

[3] Craig Chambers. Private communication.

[4] Craig Chambers. *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University Department of Computer Science, Stanford, CA, March 1992.

[5] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89 Conference Proceedings*, pages 49–70, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[6] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993. To appear.

[7] Digital Equipment Corporation. *Alpha Architecture Handbook*. 1992.

[8] The StepStone Corporation. *The Objective-C Reference Manual*. Sandy Hook, CT, 1990.

[9] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Theta reference manual. Technical report, 1994. To appear as memo of the Programming Methodology Group, Laboratory for Computer Science, MIT.

[10] 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, UT, January 1984.

[11] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA '89 Conference Proceedings*, pages 211–214, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[12] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[14] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. *The* SELF *Manual, Version 1.1*, February 1991. Unpublished manual.

[15] Douglas Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, pages 301–314, San Francisco, CA, April 1990.

[16] B. Liskov and et al. *CLU Reference Manual.* Springer-Verlag, 1984.

[17] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*, pages 79–91. Morgan Kaufmann, 1993.

[18] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice-Hall, 1991.

[19] John Plevyak and Andrew A. Chien. Incremental inference of concrete types. Technical Report UIUCDCS-R-93-1829, Dept. of Computer Science, University of Illinois Urbana-Champagne, 1993.

[20] William Pugh and Grant Weddell. Two-directional record layout for multiple inheritance. In *Proceedings of the SIGPLAN '90 Conference on Programming*

Language Design and Implementation, pages 85–91, White Plains, NY, June 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990.

[21] William Pugh and Grant E. Weddell. On object layout for multiple inheritance. *??*, pages ??–??, 1993.

[22] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*, pages 9–16, Portland, OR, September 1986.

[23] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis object-based environment, language reference manual. Technical Report DEC-TR-372, Digital Equipment Corporation, November 1985. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[24] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring '87 European Unix Systems Users's Group Conference*, Helsinki, May 1987.

[25] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3), June 1991.

[26] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241, Orlando, FL, October 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.