Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)

Bidirectional Object Layout for Separate Compilation

Andrew C. Myers
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA
`andru@lcs.mit.edu` *

Abstract

Existing schemes for object layout and dispatch in the presence of multiple inheritance and separate compilation waste space and are slower than systems with single inheritance. This paper describes the *bidirectional object layout*, a new scheme for object layout that produces smaller objects and faster method invocations than existing schemes by automatically optimizing particular uses of multiple inheritance. The bidirectional object layout is used for the programming language Theta, and is applicable to languages like C++. This paper also demonstrates how to efficiently implement method dispatch when method signatures are allowed to change in subclasses. Most current statically compiled languages require identical signatures for efficiency.

1 Introduction

Existing schemes for object memory layout and method dispatch in statically typed languages with multiple inheritance assume the use of multiple inheritance in its full generality. These schemes incur high per-object space overhead [Str87], extra dispatch cost, or a global type analysis phase [DMSV89].

These problems are addressed by the *bidirectional object layout*, a new scheme suitable for separately-compiled languages with multiple inheritance. The primary advantage of this object layout is that it supports faster method dispatch than current C++ implementations [ES90], while reducing the amount of per-object dispatch information. This paper presents precise rules for constructing bidirectional layouts and explains the dispatch code needed for typical RISC architectures.

Bidirectional object layouts can be efficiently computed because the construction rules use only local knowledge of the type hierarchy to compact the dispatch information. By requiring only local analysis, this technique can be used with separate compilation since new classes added to the system cannot invalidate the layout of existing classes. Therefore, the bidirectional layout scheme scales better to large-scale software development than other compact layout schemes requiring a global type analysis phase.

A separate contribution of this paper is an efficient implementation of method signature refinement: a type may compatibly modify the argument and return types of a method derived from its supertypes, in accordance with the usual subtyping rules [Car84]. This feature is regrettably missing from other statically compiled languages like C++ or Modula-3 [Nel91]. The technique described here allows a class to refine superclass method signatures yet retain fast method dispatch.

The bidirectional object layout has been implemented for the programming language Theta [DGLM95, LCD$^+$94, Mye94], a separately-compiled, statically-typed language which separates subtyping and inheritance. Theta allows a class to have any number of explicitly declared abstract supertypes, but only a sin-

*Web site: http://www.pmg.lcs.mit.edu/~andru/. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under grant CCR-8822158.

gle concrete superclass. This policy results from observations about the three common uses of multiple inheritance in programs, which are exploited by the bidirectional layout to generate more compact layouts:

subtyping An abstract type, or interface, combines several existing interfaces and extends them. The abstract type is a *subtype*, and the extended interfaces are its *supertypes*. Elsewhere, this use is sometimes called “interface inheritance” among “abstract classes”.

abstraction An implementation (a class) is isolated from the interface it implements, providing separate hierarchies for inheritance and subtyping. As a result, a type need not expose implementation details for subtype implementations to inherit them.

inheritance A new implementation is formed by combining and extending existing implementations. The new implementation is a *subclass*, and the extended implementations are its *superclasses*. In addition to method code, a class contains instance variables and *private methods* that are not accessible outside the class and its subclasses. In this paper, the term “inheritance” is used exclusively to mean implementation inheritance.

The bidirectional layout supports the complex abstract-type/class hierarchy shown in Figure 1 as efficiently as single-inheritance classes in C++. This common kind of hierarchy has only the subtyping and abstraction uses of multiple inheritance. In the figure, the C_i are classes, and the T_i are abstract types. Arrows are used to denote the relations “is a subtype of”, “implements”, and “inherits from”, depending on whether the two related entities are two abstract types, a class and an abstract type, or two classes. For example, T_1 is the supertype of T_2, and C_3 implements T_3. Such hierarchies are common, because they provide a layer of abstraction that separates interface and implementation.

Although subtyping and inheritance are separate, they often develop in parallel. Separate hierarchies allow us to add new implementations of any of the types in the ladder $T_1, T_2, \ldots$ without being forced to inherit from existing classes. On the other hand, an implementation of one of these types is often useful in constructing an implementation of a subtype (either direct or indirect), so inheritance often *roughly* mirrors

$$\begin{tikzcd}
 & T_1 & & \\
C_1 \arrow[ur] & & T_2 \arrow[ul] & \\
 & C_2 \arrow[ul] \arrow[ur] & & T_3 \arrow[ul] \\
 & & C_3 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 1: A common kind of hierarchy

the subtype hierarchy. For lack of a better technique, C++ programmers use true multiple inheritance and other inefficient idioms to achieve this same separation.

The bidirectional layout does better than current schemes because it implements subtyping and abstraction differently from true inheritance. It specifically optimizes lattice-like hierarchies like the one shown in Figure 1, where the types that a class and a superclass implement are in a parallel supertype relationship. Even with the use of true multiple inheritance, the bidirectional layout never imposes a penalty compared to current C++ implementations.

As in current C++ implementations, the layout of a class is determined using only information about that class and the hierarchy above it, with the result that new types and classes added to the system do not invalidate existing layouts. Changes to classes only affect their subclasses, and changes to types only affect their subtypes and the classes that implement the subtypes. Note that the supertype relation must be explicitly declared rather than being implicitly inferred from method signatures.

2 Bidirectional Layout

The bidirectional layout differs from the layouts used by many C++ implementations [Str87], though there are similarities. An object has a fixed memory layout, defined by its class. Figure 2 illustrates an object layout, which is divided in two parts (note that memory addresses increase downward within each box):

- A *dispatch header*, which is an array of pointers to various *dispatch vectors*. Each dispatch vector is itself an array of pointers to method code, indexed by small integers that are the *method indices*. The dispatch vector supports selector-table indexed (STI) dispatching [Ros88]. Each dispatch vector of

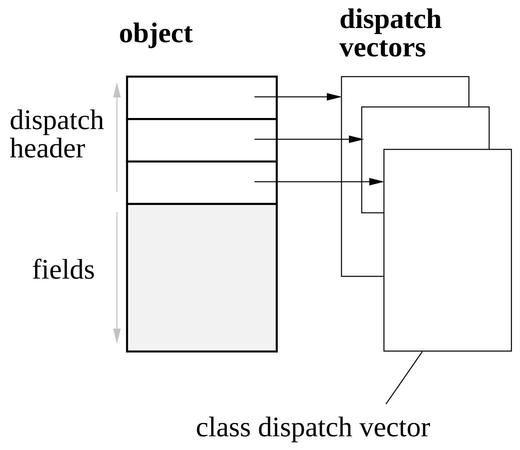


Figure 2: Bidirectional object layout

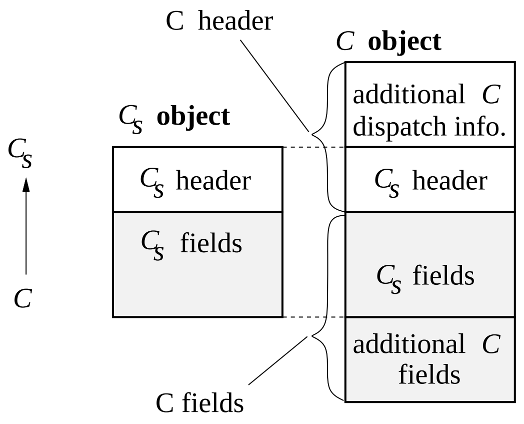


Figure 3: Class/superclass compatibility

an object specifies a different mapping from indices to methods. The last (bottom) dispatch vector, which is always present, is the *class dispatch vector*. It contains all the methods that can be called on the object. The other dispatch vectors are called *type dispatch vectors*.

Dispatch vectors are shared by all objects of the particular class that owns them, but each object has its own copy of the dispatch header that points to them. Most objects have only a single dispatch vector, so only one word is consumed by the dispatch header. With the use of inheritance or subtyping, the dispatch header layout grows backward in memory, if it grows at all.

- Fields, which contain mutable object data (usually instance variables), are located after the dispatch header. When a class inherits code and fields from a superclass, the fields of the class are placed after the inherited fields, so the fields grow forward in memory with the use of inheritance.

The bidirectional layout separates the fields and dispatch information of the object, whereas most C++ implementations intermingle them. One advantage of separating the dispatch information is that adjacent entries in the dispatch header can be merged into one entry by merging their corresponding dispatch vectors, using the rules described in Section 3.

To ensure that method code inherited from the superclass need not be recompiled, the bidirectional layout makes the layout of a class compatible with that of its superclass. Compatibility is achieved by nesting the superclass layout within that of the class, as shown in Figure 3. The dotted lines connect the layout of the superclass, C_s, to the corresponding portion of the layout of the class, C. The relationship of the two classes is indicated by the vertical arrow at the left of the figure. Because C contains a portion whose layout is compatible with C_s, inherited method code will work properly when passed a pointer to the C_s portion of the object.

In general, a layout — whether of an object, a dispatch header, or just a dispatch vector — is *compatible* with the corresponding superclass layout if the superclass layout is embedded in the subclass layout.

Both abstract types and classes have dispatch header and dispatch vector layouts. Classes also provide dispatch header data that is copied into the beginning of objects of that class and contains pointers to the dispatch vectors shared by all objects of that class.

An object reference is implemented by a pointer into the dispatch header. The dispatch vector indirectly referenced by the pointer is called the *current dispatch vector*. Any object reference has some static, declared type, although the run-time type of the object may be any of that type's subtypes. Regardless of the actual run-time object type, the method indices of the current dispatch vector agree with the method indices of the last dispatch vector of the static type's layout. Note that one dispatch vector may contain methods for several types so long as the method indices do not conflict.

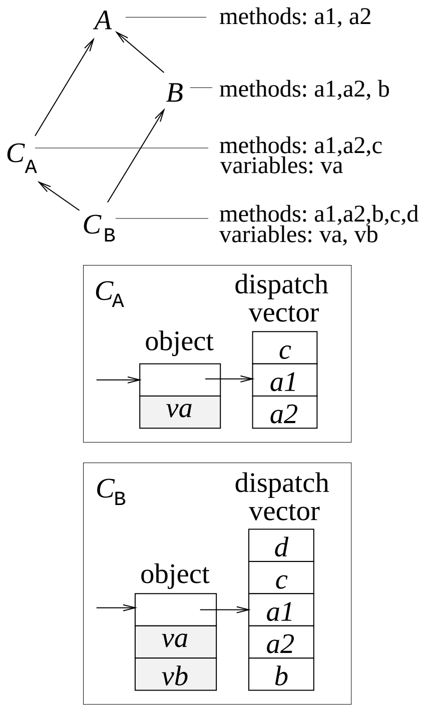


Figure 4: Bidirectional layout example

2.1 Example

Figure 4 provides examples of bidirectional object layouts for two classes, C_A and C_B. These classes implement types A and B respectively. The methods and instance variables associated with each type or class are indicated in the diagram. For example, C_A supports the methods *a1* and *a2* of its type A, and adds a private method c. It has one instance variable named *va*. The figure shows the layouts of objects and the dispatch vectors they point to. The dispatch headers for each of these classes contain a single dispatch vector, which is the class dispatch vector. Note that the C_B object is compatible with the C_A object, which is necessary because C_B inherits from C_A.

2.2 Method Indices

This section describes method dispatch and the layout of dispatch vectors. A method call is implemented by fetching an element of the current dispatch vector, at a method index determined from the statically declared type of the object. In the Theta implementation, the dispatch vector contains pointers to method code, leading to the usual three-instruction assembly-language dispatch sequence for the MIPS architecture (most RISC architectures are similar), shown in Figure 5.

```
ld t1, 0(a0)     ; load dispatch vector
ld t9, (i * wordsize)(t1) ; code pointer
jalr ra, t9           ; jump to method
```

Figure 5: STI dispatch on the MIPS

In this code sample, the register `a0` contains the object pointer. The value *i* is the method index. For any type T and method M of that type, a method index *i* identifies the proper offset within the dispatch vector. For example, in Figure 4, the method index of the method *b* in type *B* is 2.

Method indices are densely packed, which is desirable because dispatch vector space is used efficiently, yielding good cache performance. The indices of the n methods in a type dispatch vector are in the range $0 \ldots n-1$. In class dispatch vectors, some method indices are negative, and indices are in the range $-m \ldots n-1$, for some m and n. For example, in Figure 4, $m = 2$ and $n = 3$ for C_B. The use of negative method indices provides important flexibility in constructing header layouts; the reason for this will become clear later.

A type dispatch vector contains all the methods for types in a supertype chain: a series of types $T_1 \ldots T_n$, where

$$i < j \Rightarrow T_i \text{ is a supertype of } T_j$$

The first entries in the dispatch vector are for T_1, then for T_2, and so on. This ordering ensures that a subtype dispatch vector is compatible with any supertype in the chain. For example, the methods of the types $T_1 \ldots T_3$ in Figure 1 could be placed in one dispatch vector, in that order. Methods are not duplicated in the dispatch vector, so the T_j portion of the vector contains no methods that are derived from T_i, where $i < j$. Therefore, the layout of the dispatch vector is uniquely defined by the sequence of types $T_1 \ldots T_n$.

For a subtype dispatch vector to be compatible with a supertype dispatch vector, the supertype method indices must correspond to the same methods as in the subtype. This condition implies that the supertype dispatch vector must be embedded exactly at the start of the subtype vector, since any offset would require remapping its method indices.

The non-negative indices in a class dispatch vector are arranged as in a type dispatch vector. The negative indices contain methods for a series of classes $C_1 \dots C_n$, where C_{i+1} inherits directly from C_i. As in the non-negative portion, the C_1 methods are closest to zero, and indices grow away from zero. Private class methods are found only in the negatively-indexed portion, but public methods derived from types may also be there.

For example, the class dispatch vector for C_B in Figure 4 contains a negative portion with $C_1 = C_A$, $C_2 = C_B$ and a non-negative portion with $T_1 = A$, $T_2 = B$.

2.3 Object References

An object reference is a pointer to a location in its dispatch header. As described earlier, the object header is an array of pointers to different dispatch vectors. An object reference of type T points indirectly to the appropriate dispatch vector for T. The possible pointers to the object constitute *multiple views* of the object, each view corresponding to a type or types.

When object method code is running, the reference to the receiver object `self` is always a class view: a pointer to the last dispatch vector, the class dispatch vector. This requirement makes instance variable accesses fast, because the class dispatch vector never moves relative to the fields. Each instance variable resides at a fixed offset from the `self` pointer in all class views, so the variable's value can be fetched by a single indexed load instruction.

When a method is invoked on an object reference pointer that is does not point to the class dispatch vector, the pointer is adjusted by adding a small constant to it. This *pointer bumping* ensures that the method code receives the class view of the receiver. Because the actual class of the object is not known, the proper offset cannot be determined statically — a dynamic method must be used.

To produce the offset dynamically, dispatch vectors other than the class dispatch vector point to small *trampoline* procedures that bump the object pointer and then jump directly to the method code. The offset applied in the trampoline procedure is exactly the offset between the current dispatch vector and the class dispatch vector. A trampoline is shared by all objects of its class, so the total space required to store trampolines is relatively small.

Trampoline code for the MIPS R3000 is shown in Figure 6. In the figure, the register `a0` contains the

```
add a0, offset        ; offset reference
j method_code         ; jump to method
```

Figure 6: A trampoline procedure on the MIPS

receiver object reference. Because the code of Figure 5 jumps to these two instructions rather than directly to the method code, the trampoline adds an additional two-cycle penalty onto the STI dispatch cost for that processor. A similar or lesser cost should apply on most RISC architectures, since the trampoline jumps directly to the actual method code.

The trampoline technique has been used by some C++ compilers (for example, the DEC C++ compiler `cxx`, though not quite as compactly as the code of Figure 6), but seems not to be widely known. When an offset is not required, the trampoline is omitted, reducing the dispatch to just the three instructions of Figure 5.

Object pointers also sometimes require bumping when a value is viewed as a supertype of the current static type: for example, in an assignment to a variable of the supertype, or when the value is passed as an argument to a method expecting a supertype. When the dispatch vector of the supertype is incompatible with the current dispatch vector, the pointer must be bumped to produce a valid supertype view. The bump offset is a small constant that can be determined statically.

Adjacent dispatch vectors in the bidirectional layout often can be merged together to reduce the size of the dispatch header. Ideally, all the dispatch vectors are merged into the class dispatch vector, avoiding trampolines and type conversions. Thus, compact object layouts also lead to faster code.

3 Layout Rules

The following section presents the rules for constructing compact dispatch headers for the bidirectional layout. The goal of these rules is to generate compact layouts for classes that are part of typical hierarchies. No object layout scheme can generate minimal layouts without global knowledge of the hierarchy, but as these rules demonstrate, it is possible to do very well with only local information so long as the class hierarchy conforms to expectations. Even when it does not, these rules do as well as or better than rules used by current

C++ compilers.

Existing C++ compilers do not handle the hierarchy of Figure 1 well. For each level of inheritance, they add an additional word of dispatch information, making multiple inheritance an inefficient way to separate interfaces and implementations. Alternative techniques for separating inheritance and subtyping are also hampered by extra overhead. By contrast, the rules in this paper handle hierarchies of this form with only a single word of dispatch information. More complicated hierarchies involving multiple supertypes may require more dispatch vectors for some classes, but the rules here are never less efficient than currently used schemes.

3.1 Notation

The construction rules for the bidirectional object layout are described here in a compact graphical notation. The rules are essentially two-dimensional macros. This section briefly describes the notation, and provides some insight into the workings of object layout.

The dispatch headers and dispatch vectors of an object can be thought of as a two-dimensional array. The array rows are dispatch vectors and the columns are individual methods. Each row generates a word of per-object space overhead in the dispatch header. Since different dispatch vectors of the object have different lengths, the rows of the array are ragged.

Each abstract type and class in the system has a distinct dispatch header layout. These layouts are called *type headers* and *class headers*, respectively. The layout of an abstract type or class X is symbolized by placing a double box around it: $\boxed{\boxed{X}}$

The height of the box corresponds to the number of dispatch vectors in it, and to the amount of space consumed by the dispatch header in each object. The width of the box is defined as the length of the last dispatch vector in the header. Both the number of dispatch vectors in the box, and the length of the last dispatch vector, are constants for a particular type or class.

The box notation is useful because a type header is compatible with the dispatch header of each of the type's supertypes, and therefore must contain it at some offset. In the box notation, the box of the supertype must be contained within that of the subtype.

An individual dispatch vector or a portion of one is denoted by a simple box. The notation $\boxed{T_1, T_2, \ldots, T_n}$ denotes a dispatch vector that contains methods for T_1, T_2, and so on, in ascending method index order. Method pointers are not duplicated: for each T_i, the methods included in the dispatch vector are those for which there is no method pointer in the $T_1 \ldots T_{i-1}$ part.

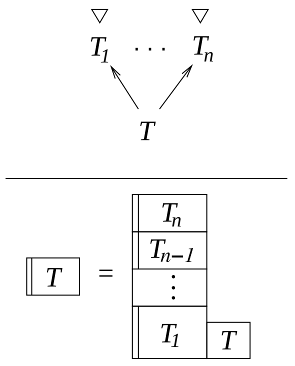


Figure 7: A simple type header layout rule

The vertical abutting of boxes means that the dispatch vectors of the lower box should be placed after the dispatch vectors of the upper box. The horizontal abutting of a dispatch header H and a dispatch vector V means that the last dispatch vector within H should have V appended to it.

This notation now can be used to express a simple but inefficient rule for constructing type dispatch headers, shown in Figure 7. This is *not* the rule actually used in the bidirectional layout; a better rule is presented later. Figure 7 is written as an inference rule. The antecedent, written above the line, indicates that the abstract type T has *direct* supertypes $T_1, T_2, \ldots T_n$. Supertypes are considered direct if they lie immediately above in the type hierarchy; indirect if they lie anywhere above. The triangles above the supertypes indicate that these types are part of an arbitrary type hierarchy extending above them.

The consequent, below the line, states that the layout for T consists of the concatenated layouts for all the supertypes, except that the last dispatch vector has been extended to include the methods of T that are not shared with T_1. The mismatch in the heights of the boxes labeled T_1 and T is intentional: the box labeled T represents a single dispatch vector, whereas the box labeled T_1 represents a dispatch header that may contain several dispatch vectors. The type T_1 is called the *primary supertype*, since its last dispatch vector is merged with that of T. The types $T_2 \ldots T_n$ are *secondary supertypes*.

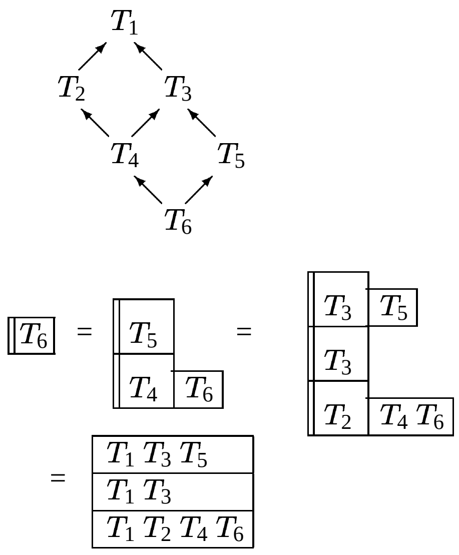


Figure 8: Inefficient dispatch header

Although this rule is not the actual bidirectional-layout rule, it has several instructive similarities. A reference to an object of type T points to the last dispatch vector in the layout. Conversion from an object of type T to any of its direct supertypes requires only the subtraction of a constant from the object pointer, which is zero in the case of T_1, since it is embedded at the end. Conversion to an indirect supertype of T requires the subtraction of a sum of these constant offsets, which is just a constant itself. Calls to any methods of T, given an object of static type T, require only the standard dispatch sequence, because all of the methods of T are present in the last dispatch vector.

3.2 Merging

The dispatch header rule of Section 3.1 works correctly but sometimes wastes space. For example, consider the type lattice shown in Figure 8. The leftmost supertype at each branch is arbitrarily considered to be primary, a convention that is followed throughout. The figure demonstrates the successive application of the rule in Section 3.1, producing the T_6 layout shown, a three-word dispatch header. Obviously, the first two dispatch vectors are compatible and can be merged, reducing the per-object overhead. Unfortunately, the simple type rule does not allow this merge.

The compatibility arises because the layouts for T_4 and T_5 contain a common T_3 piece, and can be merged. T_3 is a secondary supertype of T_4, and is exposed in the T_4 layout. Since T_3 is in a single-supertype chain above

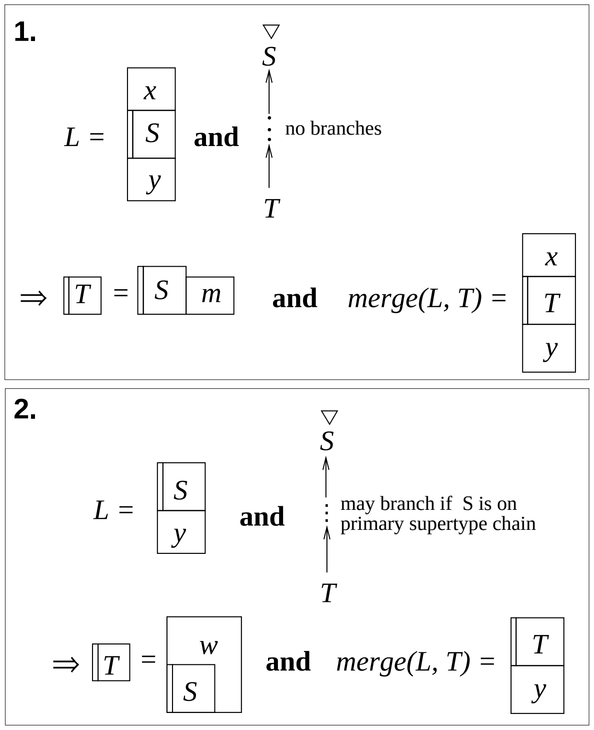
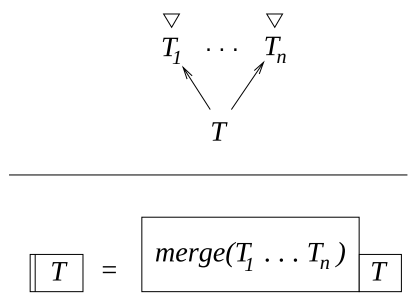


Figure 9: The two merge cases for $\leq_H$

T_5, the T_5 layout requires no more words of dispatch header space than does T_3's.

The fact that the T_5 header can be merged into the T_4 header at some offset is expressed by the $\leq_H$ predicate: $\boxed{T_5} \leq_H \boxed{T_4}$. Figure 9 provides the definition of $\leq_H$. Given a type T and an arbitrary layout L, it shows the two cases in which the T header can be merged into L. The result of the merge is compatible with L, and has the T header embedded within it. In the figure, the blocks labeled *w*, *x*, and *y* represent arbitrary pieces of dispatch header, possibly including multiple dispatch vectors. The block labeled *m* represents a part of a dispatch vector. These conventions will be followed throughout.

The first way to satisfy $\boxed{T} \leq_H L$ is if T has a supertype S whose header is exactly the same size, and which is exposed in L so that $\boxed{T}$ can be merged into the same place. For $\boxed{S}$ to be exposed in L means that nothing is appended to the end of its dispatch vectors in that layout, so that the block labeled *m* (the dispatch vector entries for methods of T that are not methods of S) has nothing to collide with. Then, since $\boxed{T}$ is compatible with $\boxed{S}$, the merge result is compatible with L. In order for the size of the T and S headers to



Rule 1: Type header layout

be the same, T and all its supertypes that are subtypes of S must have only one supertype. In other words, there is a simple chain of supertypes leading up to S.

The second way to satisfy $\boxed{T} \leq_H L$ is to expose the header of S at the *beginning* of L, and embed it at the *end* of $\boxed{T}$. Since $\boxed{S}$ is exposed at the beginning of L, and nothing is appended to its dispatch vectors, there is nothing for the *w* portion of $\boxed{T}$ to collide with, and the two layouts can be merged. In order for $\boxed{S}$ to be located at the end of $\boxed{T}$, T and all its supertypes that are subtypes of S must lie on a primary supertype chain, since, as shown later, primary supertypes are always aligned with the last dispatch vector in the type header. There may be branches in the supertype structure over T, which produce the *w* portion of the layout. Note that the dispatch vectors of T that are shared with the embedded header for S may have methods appended to them, which is why *w* extends to the right of $\boxed{S}$.

3.3 Type Header Rules

The complete rule for type header layout is shown in Rule 1. Rule 1 looks like the simple type rule of Figure 7, except that it attempts to merge the supertype headers using the function *merge*, defined by the rules in Figure 10. Like the rule of Figure 7, Rule 1 makes T_1 the primary supertype by aligning its header with that of T.

When a type has no supertypes, the result of the merge is considered to be empty, so the dispatch header for such a type contains just a single dispatch vector with the methods of T, starting from index 0. With this interpretation, Rule 1 covers all cases for type header construction.

3.3.1 The *merge* function

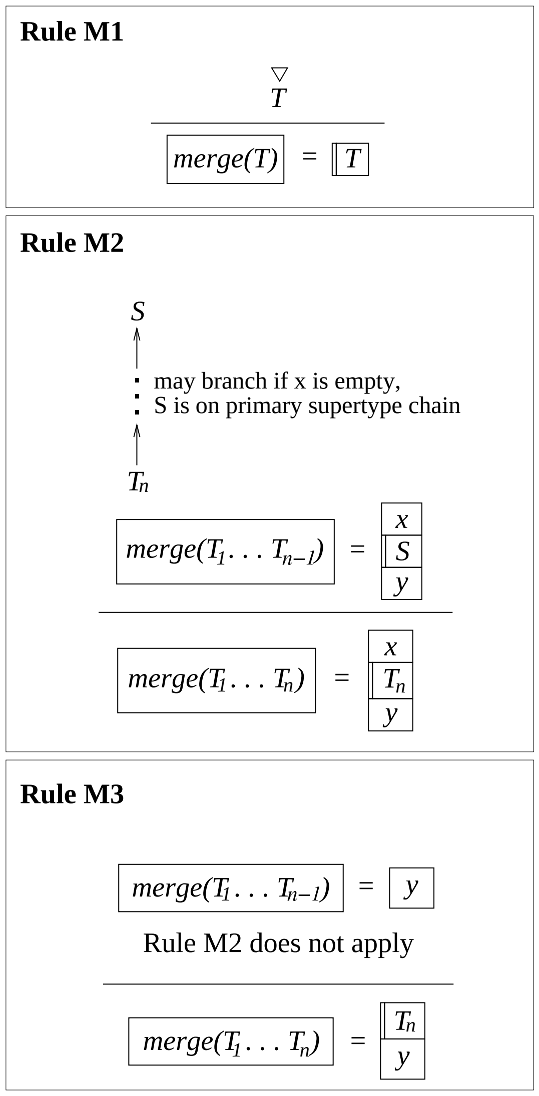


Figure 10: Merge rules

The function *merge* takes a list of types and produces a layout that merges the headers of all the types. There are three cases that the function considers, described in the rules M1, M2, and M3 of Figure 10.

Rule M1 states the obvious base case: merging a single type header yields the type header itself.

Rule M2 actually performs a merge. The antecedent to M2 captures exactly the merge cases described in Figure 9, stating that

$$\boxed{T_n} \leq_H merge(T_1 \ldots T_{n-1})$$

If *x* is non-empty, Rule M2 requires that there be no branches along the path from T_n to S, matching the antecedent to merge case 1. If *x* is empty, Rule M2 requires only that S be a primary supertype of T_n, matching the antecedent to merge case 2. The merge result from Rule M2 similarly matches the results of the two merge cases. In case 1, using Rule M2 to merge in T_n does not increase the size of the type header. In case 2, the size of the type header increases, but only by the size difference of $\boxed{T_n}$ and $\boxed{S}$.

If rules M1 and M2 fail, Rule M3 must be used: as in Figure 7, append $\boxed{T_n}$ at the start of the previous merge. This rule increases the size of the type header by the height of $\boxed{T_n}$, so Rule M2 is always preferable when it can be applied. The rules for *merge* are complete, because M3 can always be applied in the case when M2 cannot.

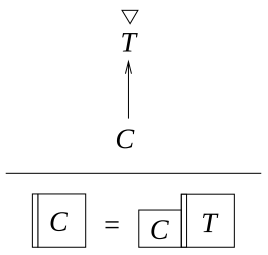
3.3.2 Type Header Rule Correctness

For the merge rules to be correct, type headers in a primary supertype chain must be aligned with each other. This alignment is guaranteed because rules M2 and M3 keep the result of the previous merge aligned with the last word of the result merge.

Also, embedded type headers must be located at a known offset from the end of the type header so that supertype conversion works properly; Rule M1 clearly preserves this property. Rule M3 keeps all embedded type headers in *y* at the same offset, and all embedded type headers in T_n are offset by the height of *y*, which is a known constant. Therefore, M3 preserves the property. Rule M2 is more complex. Clearly, M2 keeps any embedded headers in $\boxed{S}$ and *y* at the same offset. Now, consider the embedded information in *x*. Rule M2 can satisfy either of the merge cases of Figure 9. If it satisfies case 1, then $\boxed{T_n}$ has the same height as $\boxed{S}$, so the dispatch information in *x* is located at the same offset in the resulting merge. If Rule M2 satisfies case 2, then $\boxed{T_n}$ may be larger than $\boxed{S}$, but *x* is empty. Inductively, the known-offset property is preserved by the combination of Rule 1 and the merge rules.

3.3.3 Supertype Ordering

The order in which the supertypes are merged is important, because it may determine whether Rule M2 can be applied. The indices on the supertypes in Rule 1 can be assigned in an order that allows as much merging as possible. The current Theta implementation does not attempt to find an optimal ordering, since the number of supertypes is usually small. It successively picks supertypes and places them in an ordered list for *merge* using the following heuristic. At each iteration, it preferentially picks a supertype whose header can be merged into the minimal number of unpicked supertype headers. Thus, it avoids adding a header until it has added any headers that it can be merged into. Usually, there is a supertype that cannot be merged into any other supertype headers. It breaks ties by picking a supertype that can be merged by Rule M2 into the layout of the currently picked supertypes. The picked supertype is appended to the end of the ordered list, and the process repeats.



Rule 2: A non-inheriting class

3.4 Class Header Rules

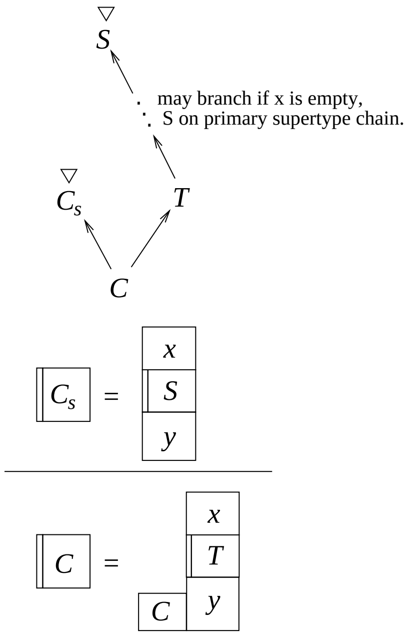
The advantages of the object layout described here are most apparent for a full class header. The kind of type lattice shown in Figure 8 is uncommon for abstract types, but similar hierarchies are frequent in a system with separate subtyping and inheritance, as shown earlier in Figure 1. Merge optimizations like that in Section 3.3 are correspondingly more important, and are used by the bidirectional layout to produce compact class headers.

In Theta, a class implements either a single type or none; it inherits either from a single superclass or from none. Classes that do not implement a type are useful because they can be inherited without fear of damaging existing objects. They can also be used for private data structures within another implementation.

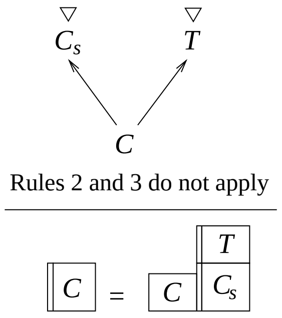
There are several cases to be considered when constructing class headers, and the following sections will enumerate them and provide the construction rules.

3.4.1 Non-Inheriting Classes

The simplest case to consider is a class that does not inherit from any superclass, shown in Rule 2. This rule optimizes the header of a non-inheriting class C



Rule 3: Merging type headers into class headers



Rule 4: Default class header rule

by making its class dispatch vector compatible with the type it implements, T.

The horizontal abutting of the two boxes, with $\boxed{T}$ on the right, indicates that the methods of C that are not found in T (the private methods) are *prepended* to the last type dispatch vector of T. The indices of the T methods do not change; the methods of C are added with negative indices starting from -1.

Note that if T has only single supertypes above it in the hierarchy (a *simple* hierarchy), then the whole class header for C will consume only a single word per object.

If a class does not inherit from a superclass or implement an abstract type, Rule 2 also applies. The $\boxed{T}$ layout is considered to be empty, so the dispatch header of such a class contains a single dispatch vector with only negatively-indexed methods.

3.4.2 Merging Type and Class Headers

The key to the efficiency of the bidirectional layout is the merging of type and class headers in type/class lattices, which is described by Rule 3. This rule computes the layout of a class C that implements type T and inherits from class C_s. The header of C_s must contain an exposed S header that T is compatible with. The C header is formed by extending the C_s class dispatch vector backward, and the embedded S header forward. The backward extension of the C_s class dispatch vector makes the layout compatible with C, and the forward extension makes it compatible with T. Both extensions can affect the same dispatch vector, which is made possible because the backward extension of the class dispatch vector cannot collide with the forward extension of S.

As in the merge cases of Figure 9, the supertype S must either be on a simple supertype chain above T, or, if it is exposed at the start of $\boxed{C_s}$ (*i.e.* x is empty), on a primary supertype chain above T. Thus, Rule 3 is analogous to the earlier Rule M2.

3.4.3 The Default Rule

Rule 4 guarantees that the class header construction rules are complete. In the rule, C is a class that implements the abstract type T, and inherits from the class C_s. The header for C is formed by concatenating the headers of T and C_s, without merging. The methods of C that are not already in the the class dispatch vector of C_s are prepended to the negative portion of the class dispatch vector. Rule 4 corresponds to merge Rule M3 and the simple type header layout policy of Figure 7: like a C++ layout, it tends to make the object dispatch overhead larger with each level of hierarchy.

When Rule 4 is applied to a class C that does not implement an abstract type, the $\boxed{T}$ portion of the layout is considered to be empty. To produce the class header for C, it just prepends the new C methods.

When T is a supertype of the type that C_s implements, the Theta implementation performs an optimization not shown in the diagram. Since $\boxed{T}$ is already embedded in the C_s header, there is no need to concatenate $\boxed{T}$ to $\boxed{C_s}$ to create the C header. The new C methods are simply prepended to the class dispatch vector.

3.4.4 Combining the Rules

When Rule 3 applies, its advantage over Rule 4 is that it does not increase the size of the dispatch header with each level of hierarchy. Type/class lattices like those depicted in Figure 1 require only rules 1, 2, and 3, and all applications of rule 1 use merge rules M1 and M2. None of these rules cause the header to grow, so the class headers of all classes in a simple type/class lattice occupy only a single word. In fact, only a single word of dispatch information is required for a class as long as the following two conditions are met:

1. No types in the hierarchy above the class have multiple supertypes.
2. Either:
   (a) The type implemented by a class is a subtype of the type implemented by its superclass, allowing merging by Rule 3, *or*
   (b) The type implemented by a class is a supertype of the type implemented by its superclass, allowing merging by Rule 4 with optimization.

This result is a significant improvement on existing techniques.

Negative method indices allow smaller headers. Without them, reasonable merge schemes can be defined [Mye94], but the size of headers in a type/class lattice is two words rather than one. The two dispatch vectors correspond to the negative and non-negative portions of the class dispatch vector.

3.4.5 Class Header Rule Correctness

Rules 2 through 4 are both complete and correct, which again can be shown inductively. For completeness, consider all possible classes. If the class does not implement a type, its header is defined by Rule 4. If the class does not inherit from a superclass, its header is defined by Rule 2. If the class both implements a type and inherits from a class, its header is defined by either Rule 3 or Rule 4, depending on whether merging is possible. Therefore, all class headers are constructible.

For correctness, embedded type and class headers must be found at known offsets from the class dispatch vector. Rules 2 and 4 clearly preserve this property. Rule 3 operates almost identically to merge rule M2, and the correctness arguments from Section 3.3.2 again suffice.

$$\begin{tikzcd}
 & & T_0 & & & \\
 & C_0 \arrow[ur] & & T_1 \arrow[ul] & & \\
 & & C_1 \arrow[ul] \arrow[ur] & & T_2 \arrow[ul] & T_4 \arrow[ull] \\
 & & C_4 \arrow[uul] \arrow[urr] & C_2 \arrow[ul] \arrow[ur] & & T_3 \arrow[ul] \arrow[u] \\
 & & & & C_3 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 11: Some complex types and classes

Bidirectional Layout

	$T_0 T_1 T_4$
$C_3 C_2 C_1 C_0$	$T_0 T_1 T_2 T_3$
C_0 *fields*	
C_1 *fields*	
C_2 *fields*	
C_3 *fields*	

Usual C++ Layout

T_0 C_0 C_1 C_2 C_3
C_0 *fields*
C_1 *fields*
T_0 T_1
C_2 *fields*
T_0 T_1 T_2
C_3 *fields*
T_0 T_1 T_2 T_3
T_0 T_1 T_4

Figure 12: Bidirectional and C++ layouts for C_3

3.5 Examples

The success of these dispatch layout rules is illustrated by the layouts for the classes in the complex hierarchy of Figure 11. Even though their layouts are computed with only local knowledge of the hierarchy, all the classes except C_3 require only a single dispatch vector. As shown in Figure 12, C_3 has just two dispatch vectors in the bidirectional layout, whereas it has five in conventional layouts. The vertical center line in the dispatch vectors for the bidirectional layout represents the separation between the negative and positively-indexed methods. Contrast these header sizes with conventional C++ layouts: classes C_1 through C_4 have 2, 3, 5, and 2 dispatch vectors, respectively. The disparity between the schemes increases with deeper hierarchies.

4 Multiple Inheritance

The material so far has been in the context of Theta, a language that does not allow multiple inheritance of code and instance variables. However, this section shows how to extend the bidirectional layout rules to support true multiple implementation inheritance. There are two approaches to supporting multiple inheritance, both of which generate alternate versions of method code inherited from some superclasses.

4.1 Recompilation

If the language allows only the instance variables of `self` to be accessed, multiple inheritance is easily implemented for a class. One of the superclasses is designated as the *primary superclass*; code is shared only with this class. The primary superclass is handled exactly as described earlier for the superclass. The secondary superclasses are treated essentially as if they were secondary supertypes, except that their fields are appended to the object layout along with the fields of the class.

Since the secondary superclass fields do not begin after the class dispatch vector, the offset to them is different than in the superclass. A new version of secondary superclass methods is compiled or linked to use the correct offset for this class.

Recompilation is easy, and produces compact objects with fast dispatching. However, it does not work if instance variables of objects other than `self` can be accessed, because the code accessing the variable cannot determine the proper field offset. Also, this solution causes some code duplication, since a class is recompiled for each distinct field offset used.

4.2 Dynamic Field Offsets

A more flexible technique for supporting multiple inheritance is to determine the location of fields dynamically. This technique produces at most two versions of method code, and allows access to instance variables of objects other than `self`.

The location of the fields is provided by placing the correct field offset in the class dispatch vector, exactly as if the offset were a private method pointer. Many implementations of multiple inheritance have used intra-object pointers for this purpose; however, dynamic field offsets take no space per object, and are usually just as fast. Since dispatch vectors are shared, field offset loads are less likely than intra-object pointers to cause cache misses. The MIPS code to place the starting address of the fields into the register `t3` is shown in Figure 13. Note that this computation may be amortized over all field accesses to a particular object within a method call.

```
ld t1, 0(a0)      ; load dispatch vector
ld t2, fields(t1) ; load field offset
add t3, a0, t2    ; start of fields
```

Figure 13: Field access setup code

Dynamic field offsets can be used to extend the recompilation scheme of Section 4.1 so that it supports accesses to instance variables of objects other than `self`. These accesses use the dynamic field offset, since the correct field offset cannot be determined statically. When instance variables of `self` are accessed, the correct offset is known, as before.

A more space-efficient technique is for *all* code to access instance variables using the field offset. This approach has the advantage that all class methods can be inherited without recompilation; it has the disadvantage that accessing the instance variables of `self` requires the code of Figure 13.

A reasonable compromise is to generate at most two versions of the class code. In both versions of the code, instance variables of objects other than `self` are accessed through the dynamic field offset. The versions differ in how they access the instance variables of `self`.

The first, fast version is used for objects of the class itself and for subclasses that inherit from the class in a primary superclass chain. Since the field offset is known statically, instance variables are accessed by an indexed load, as in the basic bidirectional scheme.

The second, generic version accesses all instance variables through the dynamic field offset code of Figure 13, and is suitable for inheritance as a secondary superclass method. This second version is particularly appropriate for mix-in classes.

5 Method Signature Refinement

The most general subtype rules allow subtypes to widen argument types and narrow result types [Car84]. This process is sometimes called method signature refinement. Because implementing these rules efficiently is difficult, separately-compiled languages like C++ and Modula-3 have a more restrictive subtype rule: they require that argument and return types stay the same in subtype methods. However, the Theta implementation demonstrates that refined method signatures can be efficiently implemented even in separately-compiled languages.

The subtype rules are difficult to implement because objects have multiple views (see Section 2.3), a necessary feature of systems with fast dispatch and separate compilation, such as Theta or C++. Multiple views are also useful for treating primitive values (*e.g.*, integers) as first-class objects in a system with a universal supertype.

To see why multiple object views make signature refinement problematic, consider two abstract types T and T_s, where T_s is a secondary supertype of T. An offset is required to convert between the T and T_s views. Suppose that the supertype T_s has a method `identity` that returns a T_s, but that in T, `identity` is declared to return a T. This hierarchy is sound, but creates practical implementation difficulties. Let `x` be a variable of declared type T_s that actually refers to a T object. Calling `x.identity()` yields a T, but the caller expects a T_s — a different view of the object.

The solution to the puzzle is straightforward: the compiler assigns multiple method indices to the method, one for every type with a distinct signature for that method. Let m_s be the method index of `identity` in T_s. Because T declares a new signature for `identity`, it introduces a new method index m for calls that use the new signature. The compiler sends method calls to the appropriate method index based on the static type of the receiver object, so the presence of two distinct dispatch entries is never visible to the programmer.

Classes that implement T_s do not need to know about T and only have a m_s entry, so the locality principle is preserved. Classes that implement T are really implementing the T version of the method, so the m entry points to the method code. The m_s entry points to automatically generated trampoline procedure, whose code bumps any arguments and calls the T version of the method, then bumps the return values appropriately. This trampoline makes the implementation of the method look as if it had the T_s signature. On the MIPS R3000, the bumping of arguments and return values imposes a one cycle overhead per adjusted value, since all adjustments are constant additions. In many cases, a trampoline procedure would be required anyway to bump `self`, and the two trampolines can be merged.

Note that a single dispatch vector can contain multiple versions of the same method, if no change of view is required among the types supporting the method, and its signature differs in them. This situation is not a problem, since the method index is chosen according to the static type of the receiver object.

This technique applies to almost any multiple-view system with inheritance, such as the bidirectional object layout. It would also allow efficient implementation of refined method signatures in C++ if the language permitted.

Overhead	Bidirectional	C++
1	44	23
2	1	17
3	0	4
4	0	1

Table 1: Theta library object overheads

6 Empirical Results

The bidirectional layout efficiently handles hierarchies with separation of inheritance and subtyping. Two class libraries were examined to determine how the bidirectional layout works in practice and whether existing code conforms to the assumptions that it is based on.

6.1 Theta Library

The standard Theta library comprises 16 classes and 19 abstract types, and several other small libraries are under development. Considering the two largest libraries, one implementing the OO7 database benchmark [CDN93], and another implementing the Labbase database [RSG95], there are a total of 51 abstract types and 45 classes. Table 1 shows the number of words of per-object dispatch overhead incurred by these classes with the bidirectional layout and with a standard C++ layout. Each entry in the table shows the number of classes that have a particular number of words of dispatch overhead. Of these classes, only one requires more than a single dispatch vector with the bidirectional layout. It occupies a position in the hierarchy very similar to C_3 in Figure 11, and has two dispatch

Overhead	Bidirectional	C++
1	399	372
2	17	24
3	8	17
4	2	6
5	0	3
6	0	1
$\geq$7	0	3

Table 2: InterViews 3.1 object overheads

vectors.

The per-object space overhead is clearly lower for the bidirectional layout. Per-class space overhead is also less: the bidirectional layouts have a total of 46 dispatch vectors, compared to 73 in the C++ layout. The bidirectional dispatch vectors are larger on the average than the C++ dispatch vectors, but they do not repeat any information that is not also repeated by the C++ dispatch vectors. Since only one trampoline routine needs to be generated for the bidirectional layout, total space usage is smaller. However, per-class space usage is less important than per-object space usage, since the total number of objects in any significant program is likely to be much larger than the total number of classes in use.

6.2 InterViews Library

An example of larger-scale object-oriented programming is the InterViews 3.1 toolkit [LCV88], written in C++ with multiple inheritance. Including component libraries, it comprises about 426 public classes. It has evolved along with C++, so the code contains some historical artifacts: most of these classes are written in a single-inheritance style with no separation of inheritance and subtyping, and have only a single word of dispatch information even with the C++ layout. Nevertheless, perusal of these classes reveals that the bidirectional layout would make many of them more efficient, using the multiple-inheritance extension of Section 4.2. The results are shown in Table 2.

6.2.1 Space Usage

With the bidirectional layout, every class is as compact as in the layouts generated by the DEC C++ compiler `cxx` (other compilers are similar), and 46 classes are more compact, some by as much as 4 words per object. Additional classes would be more compact if the uses of inheritance were “virtual” — the current classes sacrifice inheritability for performance, so the results here are generous to C++.

The total number of dispatch vectors is 465 for the bidirectional layout, and 537 for C++. Assuming that dispatch vectors all contain x words, 66x trampoline procedures must be generated for the bidirectional scheme versus 111x for C++ (Non-trampoline techniques require even more space). The total space used by shared dispatch information in the two layout schemes is 597x words for the bidirectional layout and 759x words for the C++ layout. Again, per-object space overhead is likely to dominate the dispatch vector overhead in real programs.

6.2.2 InterViews Abstraction Techniques

The attempt to separate inheritance from subtyping is a major source of inefficiency in the InterViews classes. About 30 classes (mostly members of the core InterViews library) hide their implementation in a pointer to a private class. This mechanism is actually less powerful than the separation provided by Theta. The use of private implementation classes suggests that the hierarchy of Figure 1 is common or at least desirable, as assumed.

Unfortunately, the private implementation pointer imposes even more space overhead than multiple inheritance. In addition to an extra object pointer at every level of inheritance, the fragmentation of the object into two pieces imposes at least another word of overhead per object with commonly-used memory allocators. This fragmentation overhead has not been added to Table 2.

It is interesting to note that the InterViews classes use multiple inheritance mostly for abstraction. Among the ten uses of multiple inheritance in the public InterViews classes, there is only one use of true multiple inheritance; the rest permit one class to implement several unrelated interfaces. For example, the classes for which the bidirectional layout generates a three-word dispatch header all implement three unrelated interfaces: `Handler`, `MonoGlyph`, and `Observer`. In a separate-compilation environment where densely-packed method indices are compiled into the code, a three-word dispatch header is minimal for these classes.

7 Related Work

The bidirectional layout can be seen as a way of tuning the basic C++ object layout approach [Str87] to work well with current architectures and programming practice. Almost all the optimizations defined here can be applied to C++ compilers in a straightforward manner.

Much work on object layout has focused on assigning non-conflicting indices so each class has a single dispatch vector [DMSV89]. The resulting dispatch vectors may be *sparse* and contain multiple holes, so compressing them is a major concern [PW90, Dri93].

There are two important problems with these techniques: first, they are not compatible with separate compilation. Adding a new class or classes to the system can require recomputing method indices for other portions of the system, since the entire type hierarchy must be analyzed to select method indices. Second, in a large system, the complex (usually coloring-based) algorithms can require a lot of time; times on the order of an hour have been reported for the Smalltalk-80 hierarchy [AR92]. For many systems, these techniques are impractical, and a scheme based on fast local rules is preferable.

Rose [Ros88] investigates how to perform the method dispatch operation and settles on *fat dispatch tables*, where method code is placed directly in the dispatch vector. Fat dispatch vectors will work with bidirectional layout, but seem unlikely to improve on the standard dispatch sequence — the sequences usually take the same amount of time, and fat entries take a performance hit when the method is too large to fit, because an extra jump instruction is required. In addition, fat entries may hurt cache performance.

The implementation techniques for multiple inheritance described in Section 4 involve compiling different versions of method code for different inheriting contexts, a technique similar to the production of specialized code in SELF [CUL89, Cha92] — though the technique is used there to solve a more general problem, since SELF is not statically typed. In the SELF implementation, different versions of the code are produced for every inheriting class, so code sharing is minimal and code duplication can become a problem. The two techniques suggested in Section 4 preserve code sharing better. The more general technique produces at most two versions of the inherited code, and thus does not suffer from excessive code duplication.

8 Conclusions

This paper has presented the bidirectional object layout, a new layout scheme that produces compact objects with fast method dispatch. The scheme works for statically typed, separately compiled object-oriented languages with explicit subtype declarations, an important category that previous object layout work has not fully explored. The rules for constructing new objects depend only on local information about the type hierarchy, so existing object layouts are not invalidated by extensions to the hierarchy. Therefore, the bidirectional layout scheme scales better to large-scale software development than other schemes requiring a global type analysis phase.

The object layouts produced by this scheme are as compact or more compact than those produced by current compilers, and dispatch is as fast or faster on stock hardware. Typically, the dispatch information is compacted into a single dispatch vector, and method dispatch is as fast as a single-inheritance implementation of C++.

There are two key insights behind the scheme: first, the three different uses of multiple inheritance are treated differently in the scheme, allowing dispatch information to be compacted in ways that are not possible for full multiple inheritance. Second, the object layout is reordered, separating the dispatch information from the fields of the object. The dispatch information can then be merged by the set of formal rules that have been presented here.

Perusal of existing code shows that existing C++ code also separates the functionalities of subtyping, abstraction, and inheritance, and that the bidirectional layout will support many existing hierarchies more efficiently.

Finally, the paper has shown how full method subtyping rules with signature refinement can be supported efficiently in the bidirectional scheme and in most existing multiple-view systems.

Acknowledgements

I would like to thank Kavita Bala, Miguel Castro, Dawson Engler, Umesh Maheshwari, and the referees for their excellent comments on this paper, and

especially Barbara Liskov for her support and advice on this paper and the thesis that led to it.

References

[AR92] Pascal André and Jean-Claude Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA '92 Proceedings*, pages 110–126, Vancouver, BC, Canada, October 1992.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, LNCS 173,* pages 51–68. Springer-Verlag, 1984.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 12–21, Washington, DC, May 1993.

[Cha92] 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.

[CUL89] 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.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95 Proceedings*, Austin, TX, October 1995.

[DMSV89] 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.

[Dri93] Karel Driesen. Selector table indexing & sparse arrays. In *OOPSLA '93 Proceedings*, pages 259–270, Washington, DC, September 1993.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[LCD$^+$94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Lab for Computer Science, Cambridge, MA, February 1994. Also available at `http://www.pmg.lcs.mit.edu/papers/thetaref/`.

[LCV88] Mark Linton, Paul Calder, and John Vlissides. *InterViews: A C++ Graphical Interface Toolkit*. Technical Report 358, Stanford Computer Systems Laboratory, July 1988.

[Mye94] Andrew C. Myers. *Fast Object Operations in a Persistent Programming System*. Technical Report MIT/LCS/TR-599, MIT Laboratory for Computer Science, Cambridge, MA, January 1994. Master's thesis.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[PW90] 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.

[Ros88] John R. Rose. Fast dispatch mechanisms for stock hardware. In *OOPSLA '88 Conference Proceedings*, pages 27–35, San Diego, CA, October 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[RSG95] S. Rozen, L. Stein, and N. Goodman. Labbase: A database to manage laboratory data in a large-scale genome-mapping project. *IEEE Computers in Medicine and Biology*, 1995. To appear.

[Str87] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring '87 European Unix Systems Users's Group Conference*, Helsinki, May 1987.