A Fast and Efficient Method Dispatching for Statically Typed Multiple Inheritance Object-Oriented Languages

Keung Hae Lee and *Dennis Kafura*

TR 89-40

A Fast and Efficient Method Dispatching for Statically Typed Multiple Inheritance Object-Oriented Languages

Keung Hae Lee Dennis Kafura

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

ABSTRACT

Inheritance is an invaluable mechanism for object-oriented programming. The benefits of inheritance have been well recognized over the last few years. However, these benefits typically come at the expense of run time overhead in time and space. While an efficient late binding mechanism based on indexing has been popularly used in supporting single inheritance, a mechanism for multiple inheritance which can provide a comparable efficiency has been sought. In this paper, we describe a late binding mechanism for statically typed object-oriented programming languages with multiple inheritance. Our technique, based on the partitioning of a multiple inheritance hierarchy, is a significant improvement in both space and time over other existing techniques. The fast and efficient late binding mechanism called *hierarchy partitioning* is presented. An analysis of the predicted performance of the technique and a detailed comparison with other related work are also provided.

1 Introduction

Three significant benefits of inheritance in object-oriented programming are reusability, code sharing, and subtyping. The cost of supporting inheritance is the overhead associated with late binding[1] of methods. Late binding of methods is forced by the support of redefinitions of inherited methods and subtyping. While an efficient mechanism is available for the implementation of single inheritance, a comparable technique for multiple inheritance has been sought. Most reported techniques are either too inefficient or provide only partial support.

[1]Late binding is also commonly called dynamic binding, run-time binding, and method dispatching.

This paper describes an efficient late binding mechanism for statically typed object-oriented languages with multiple inheritance. The mechanism is a generalization of the virtual function table approach pioneered by single inheritance in C++ [Stroustrup 87A]. This new technique, called *Hierarchy Partitioning*, provides an efficient mechanism for the implementation of multiple inheritance of a statically typed language like C++. Its efficiency is close to that of single inheritance C++ in both space and time.

The reader is assumed to be familiar with object-oriented programming languages, preferably, a statically typed object-oriented language like C++ [Stroustrup 86] or Eiffel [Meyer 88]. While we are using the subclassing model of C++, it should be noted that the techniques described in this paper are general and not limited to C++.

The remainder of the paper is structured as follows. Section 2 briefly describes the common index technique, a fast binding mechanism for single inheritance. Section 3 examines the possibility of extending the common index technique to multiple inheritance. Several problems associated with the approach will be noted. Section 4 introduces the hierarchy partitioning technique by an example. Section 5 describes how the new technique is used in program translation. Section 6 provides a more rigorous description of the hierarchy partitioning. The expected performance of hierarchy partitioning is analyzed in Section 7. Section 8 is a comparison with other proposed mechanisms.

2 Fast Method Dispatching for Single Inheritance

The fast method dispatching technique for single inheritance described in this section is used in C++ [Stroustrup 87A]. In single inheritance, method dispatching is done by indexing into a run time table, called the *method table*. Figure 1 shows a single inheritance hierarchy and the associated method tables. A method table is created for each class at compile time. The method table of a class is an array of addresses of methods which are available to its instances and can be redefined by its subclasses. The method table of a subclass extends its parent's method table with additional entries for new methods defined by the subclass. A new method which redefines an inherited method uses the entry of the method being redefined. For example, the method table of class C in Figure 1 contains only one entry for f(), which points to f() defined by class C.

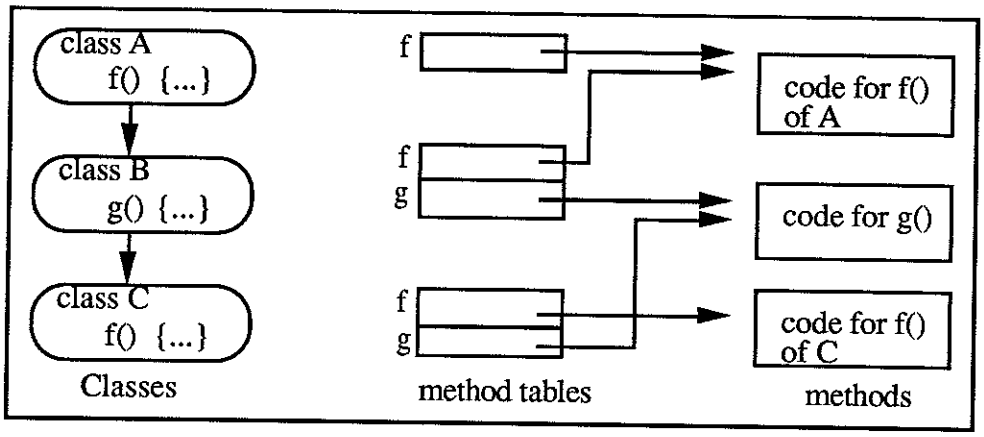

Figure 1 - Method tables of classes in single inheritance hierarchy

The compiler generates an index for each method invocation. At run time, the index is used to locate the address of the method to be executed in the method table of the receiver object. Note that this fast binding via an indexing operation is enabled by the property that two methods having the same name use a common index.

Static Binding

A strategy for detecting a method which is allowed to be redefined by a subclass is to use an explicit keyword for such a method in the definition of a class. This technique, "virtual function", has been successfully used in C++. The programmer specifies the keyword "virtual" in the declaration of a method that may be redefined by a subclass. Static binding is used for all non-virtual functions. Static binding can also be used for non-virtual functions with multiple inheritance.

Since an implementation of multiple inheritance can also use static binding for non-virtual methods, the following discussion of multiple inheritance will be concerned only about binding of virtual methods.

3 Method Dispatching for Multiple Inheritance[1]

The common index technique which enabled the fast method dispatching for single inheritance does not work for multiple inheritance. Consider the classes shown in Figure 2. A redefining method reuses the entry occupied by the redefined method as in single

[1]Semantic issues such as resolving name clashes in a subclass are discussed in other papers [Lee and Kafura 90A, Snyder 86]. For the purpose of this paper, we simply assume the inheritance model being used resolves such issues.

inheritance. Note that method b has index 0 in B's table while its index in C's table is 1. When class B is compiled, it cannot be known what b's indices will be in its subclasses. Hence, the indexing technique is not directly applicable to multiple inheritance.

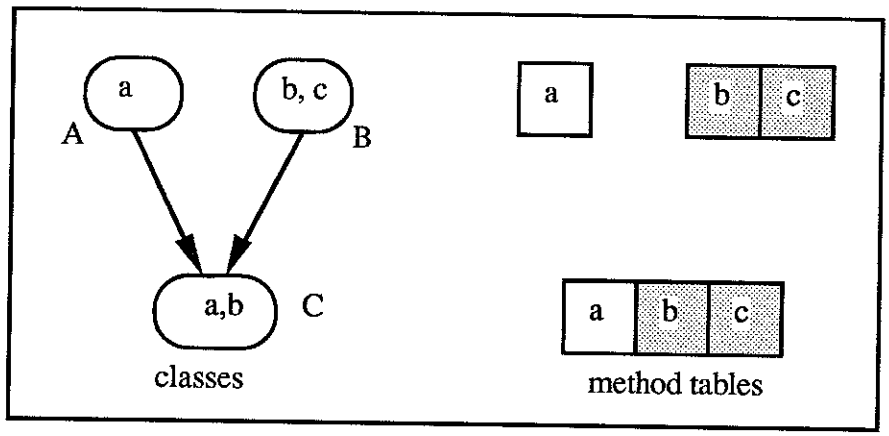

Figure 2 - Method Table with Shared Entries

Figure 3 is an attempt to fix the problem described above. A separate entry is used for each different definition in a method table. For example, the method table of C is a combination of the method tables of its superclasses, A and B, with the additional methods defined by C itself. Unfortunately, this scheme has a problem, too. A situation which is similar to the previous example occurs. Observe that b is the first entry in B's method table, but is the second and fifth entry in C's table.

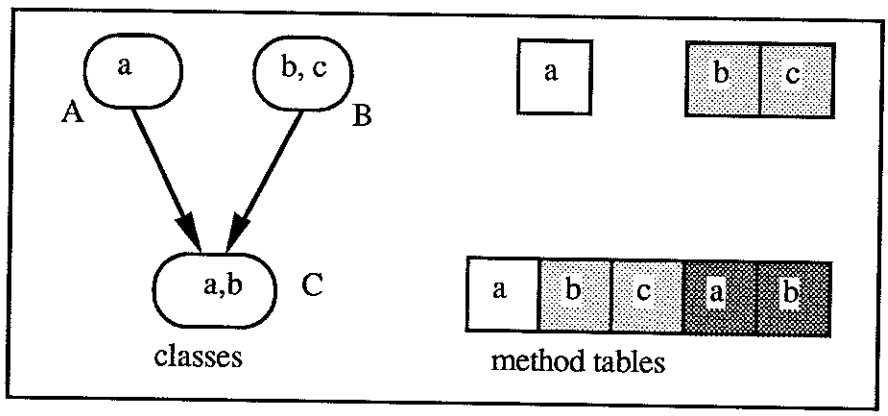

Figure 3 - Method Table with Separate Entries

Another possible solution is to modify the way a method is addressed in a method table. For the method table of a class, the compiler computes the offset value for each superclass's table component included in the method table. Each method dispatch needs to adjust the table base address before indexing by a method offset.

While this scheme is workable, it has several disadvantages which renders it undesirable. First, the scheme is inefficient in space. Second, the extra addition operation required for dispatching each method slows the binding process at run time. These two run time overheads are significant enough to warrant a new solution. A description of our new approach follows.

4 Method Dispatching with Hierarchy Partitioning

In this section, we briefly describe the new method dispatching technique called *hierarchy partitioning* by an example. A more detailed description of the technique will be provided in Section 5. An example multiple inheritance hierarchy is shown in Figure 4. Consider the method table of class I. The method table created by the separate entry scheme described in Section 3 and the one created by our scheme are shown in Figure 5.A and 5.B, respectively. For the example used here, the new scheme uses only half of the space required by the separate entry scheme. We now describe how the method table of class I is created by our scheme.

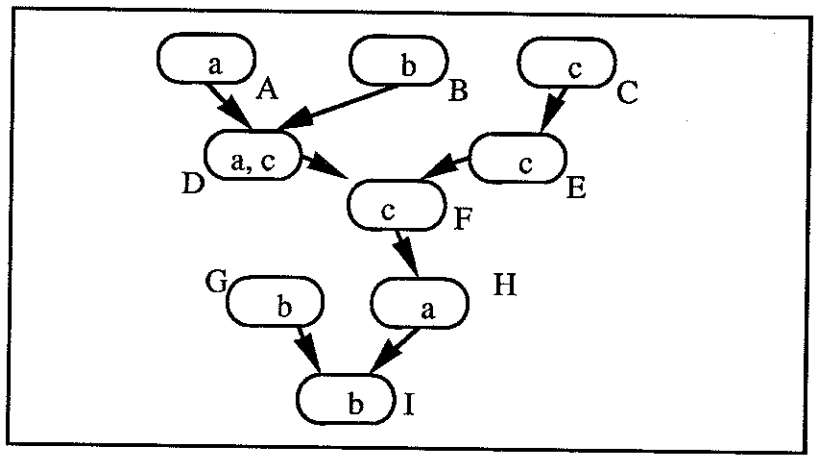

Figure 4 - A Multiple Inheritance Hierarchy

In order to create the method table of class I, we first partition the multiple inheritance hierarchy into single inheritance hierarchies. The particular hierarchy shown in Figure 4 is divided into four partitions, shown in Figure 6. The next step is to create a method table for each of classes I, H, B, and E, which are the lowest class in each partition. Each partition is considered to be an independent single inheritance hierarchy which has no connection to other partitions.

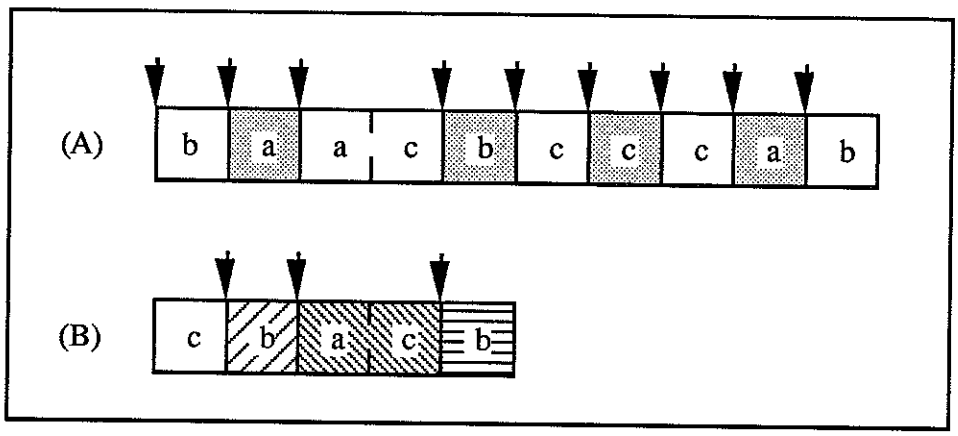

Figure 5 - Method Table for class I

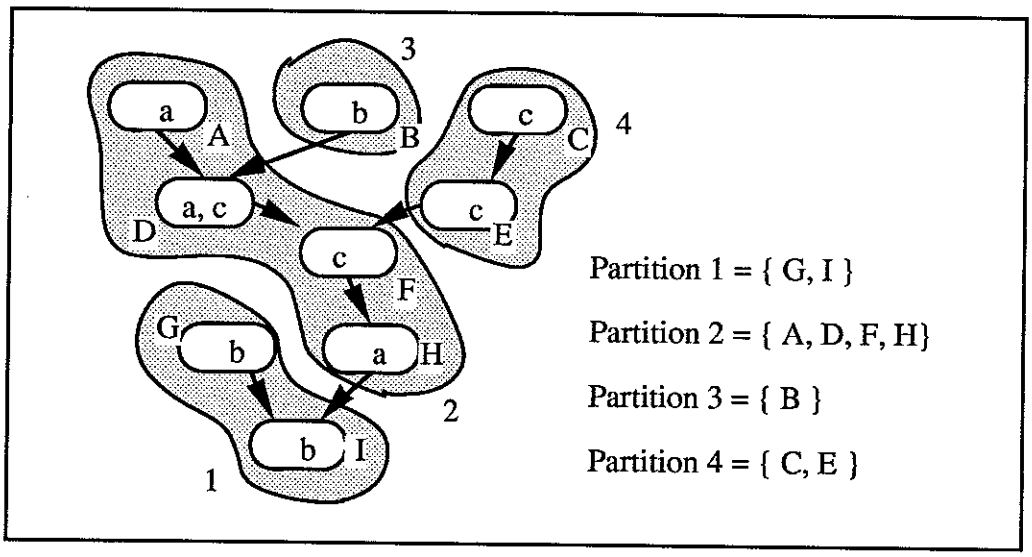

Figure 6 - Partitions for Class I

Figure 7.A shows the method tables for these four partitions. The method table for each partition is called a *component table*. Note that in each component table, methods with the same name shares the same slot as in single inheritance. For example, the component table for partition 1 contains only one entry for method b while b is defined by both G and I.

The method table of class I is constructed as a composition of the component tables. Figure 7.B shows the final method table of class I, which contains four component tables, namely, those for partitions 1, 2, 3, and 4, in that order from right to left. The offset of the component table for a partition in the final method table is called the *table offset* for the

classes in the partition. In Figure 7, class C and E have table offset 0[1] while classes A, D, F, and H have table offset 2.

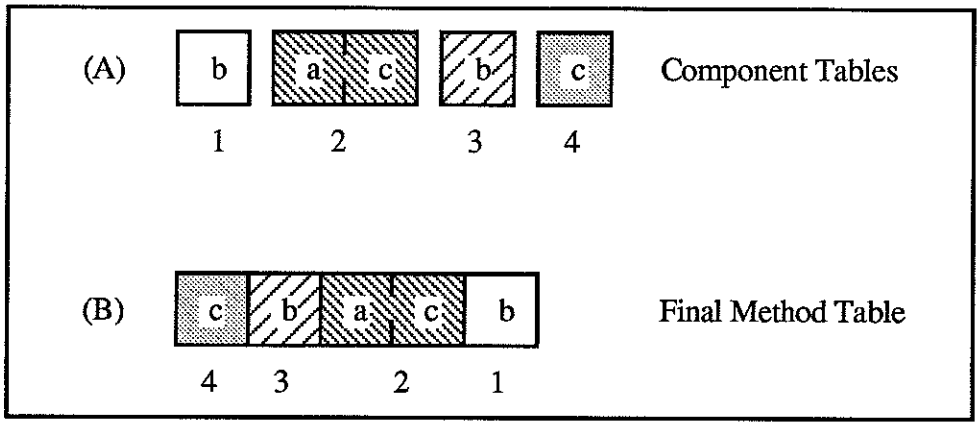

Figure 7 - Method table of class I

5 Translation with Hierarchy Partitioning

The offset of a method defined by a class is computed as its offset in the component table for the partition of which the class is a member. For the example shown in Figure 4, the method a in classes A, D, F, and H has index 0 while method c in A, D, F, and H has index 1. In the method dispatching at run time, the classes belonging to the same partition will use the same table offset for indexing into the method table. The entry for the method f stored in a method table x is located at

starting location (x) + method table offset (f) + index(f)

In our dispatching scheme, the operation of adding the method table offset to the starting location of a method table is performed during an assignment operation rather than during each method dispatch. This shift in the time at which an adjustment of method table address is performed allows the overhead of an addition incurred at each method dispatch to be amortized over the number of method invocations applied to the same object. Furthermore, the class hierarchy partitioning requires the addition operation only for a small percentage of assignments. Before presenting a more detailed description of the method dispatching

[1]We follow the convention that the index of the first entry of a table is zero.

technique, we describe the model of how a method table created by the partitioning technique is used in method dispatching at run time.

The run time data structures associated with an object is shown in Figure 8. In our scheme, an object variable contains a reference to a data structure called an *object descriptor*. An object descriptor is a data structure which consists of two addresses. One is the address of the method table of the class of which the object is an instance, and the other is the address of the actual data of the object. While efficient binding of a reference to an instance variable is also a difficult problem, this paper focuses only on the binding of methods. Binding of instance variables is discussed in a separate paper [Lee and Kafura 90A].

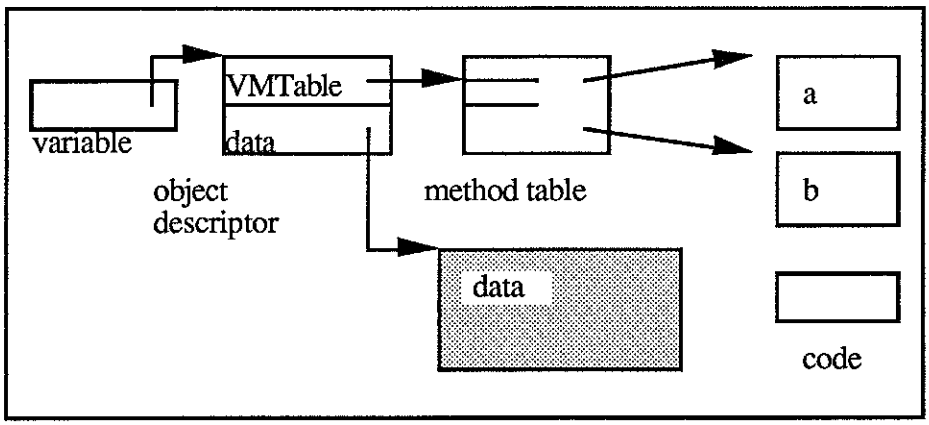

Figure 8 - Run time data structures associated with an object

In the following sections, we consider translation of language constructs which are relevant to the implementation of inheritance. Translation of assignments and method invocations will be described.

5.1 Translation of an assignment

We will consider only an assignment where both the source and the target are variables. The case when the source is an object rather than a variable is identical.

```
b = c;
```

If b and c are variables of the same type, the translation of the assignment is straightforward. The execution of this statement will cause the variable b to become another the reference to the object currently named by the variable c. Both b and c will become

references to the same object descriptor after the instruction. In this case, it is sufficient to translate the assignment into an instruction which copies the value stored in variable c into variable b at run time.

A more interesting situation occurs when b and c have different types. Let B and C denote the class of b and c, respectively. Then, the subtyping rule of an object-oriented language dictates that B should be a superclass of C. The situation is further divided into two cases depending on whether B and C are members of the same partition created during the hierarchy partitioning for the class of which the object named by variable c is an instance. If B and C are members of the same partition, the assignment is translated into a copy instruction as in the case when b and c are of the same type. This is because classes B and C have the same method table offset in the method table to be used. Since the field (VMTable) of the object descriptor already contains the correct address for the starting location of the method table component, variables b and c can share the same object descriptor. After this statement is performed, both b and c will be the references to the object descriptor which previously named by c.

If B and C are members of different partitions, the value of VMTable field in the object descriptor needs to be adjusted, and so the object descriptor for variable c cannot be used for variable b. In order to compute the starting location of the method table component for class B, we need to know the distance between the offset of the method table component for class B and the one for class C in the method table of the object which is currently named by the variable c. While the distance between two method table offsets can be computed at compile time for any two classes in a class hierarchy, the problem is how to know the actual class of the object currently named by the variable c. Due to the subtyping of an object-oriented language, the object currently named by c can be an instance of not only class C but also any of its subclasses. Hence, the actual type of the object named by a variable cannot be known until run time.

Fortunately, a theorem, which we will introduce later, lets us use the partitions of the class hierarchy for constructing the method table of class C instead of the the actual class of the object under consideration. Hence, the distance between the method table offset of class B and that of class C is completely determined at compile time. Furthermore, note that the overhead of computing this value is small since a table of method table offsets can be easily created during the process of partitioning the class hierarchy.

The translation of the assignment statement requires that a new object descriptor be created for variable b. The descriptors for b and c will have different values for the starting location of method table indexing. Let d denote the distance between the method table offsets of classes B and C. Then, the following actions need to be taken at run time.

```
b->VMTable = c->VMTable + d;
b->object = c->object;
```

5.2 Translation of invocations

We now consider how to translate a method invocation. Four different kinds of method invocations are found in existing object-oriented languages. The first and probably the most difficult case is the invocation of a method on a variable. The second is when a method is applied to the pseudo variable *self*. The third is an invocation of method on the pseudo variable *super*. The fourth case is when a method being invoked is qualified with a class name. Among these, the last case may be viewed as a variation of the third case. A description of a translation scheme for each of these follows.

Invocation on a variable

```
x.f();          // invoke f() on x.
```

Let x be a variable of class X. The above expression is an invocation of method f on the variable x, more precisely, on the object currently named by variable x. The code to be executed is determined by the type of the object named by x. Since the type of the object is not known until run time, it is not possible to statically bind the invocation to any specific definition of f() at compile time. Hence, the binding of f() must be delayed until run time when the actual object named by x becomes known, i.e., late binding is necessary.

With our method, locating the code of the method being invoked takes only a single indexing into a method table. On encountering x.f(), the compiler computes the offset of f() within the method table of X. Let d denote the offset of f(). At run time, the address of the code for f() is located at index d from the point whose address is stored in x->VMTable. By the time the above expression is executed, the object descriptor pointed to by variable x will have the correct address of the method table component for X within the method table of the actual object currently named by x. Hence, the above expression is translated into a

call to a function whose address is located using the value of x->VMTable as the base address and the offset of f() within X's method table as the offset in a relative addressing. The action can be expressed in C as follows:

```
(*x->VMTable[d])();[1]
```

Invocation on self

```
self.f()
```

While the pseudo variable self has a variety of names in different languages, every language supporting inheritance has the concept of self. In most object-oriented languages, a method invocation on self is abbreviated to a call to the method without an explicit specification of a keyword corresponding to self. In our scheme, self is a reference to an object descriptor for the sender object.

Suppose that the above expression was encountered in the definition of method u() in class X. The expression is translated into a call to the function whose address is located in the method table of self via a relative addressing (indexing) which uses self->VMTable and the offset of f() in X's method table as the base address and the offset , respectively. Let d denote the offset of f(). The invocation is translated into

```
(*self->VMTable[d])();
```

Invocation on super

```
super.f()
```

This is an invocation of f() defined by a superclass of the sender object. The method to be executed is the one inherited from a superclass. The rules set forth by the inheritance model being used determine the parent which appears in the path to the class containing the definition of f() being inherited. The compiler needs to find the offset of f() in the method

[1]Translation of an invocation involves binding the receiver object. Hierarchy Partitioning can also be used for efficient binding of instance variables. In order to focus on method dispatching we momentarily ignore the issue of binding variables. The use of Hierarchy Partitioning for variable binding will be discussed later when we compare it with the C++ implementation scheme.

table of the parent class at compile time. At run time, the code for f() is located in the method table of the parent class. The above expression is translated into an indexing operation in which the address of the method table and the table offset are known at compile time. Let M, T, d denote the addresses of the method table of the parent class, the table offset, and the index of f(). Then the translation of the above invocation looks like

```
(*(M+T)[d])();
```

Invocation qualified with a class name

```
B::f()
```

A method invocation qualified with a class name is found in C++ [Stroustrup 86] and Trellis/Owl [Shaffert et al. 86]. The above expression is an invocation of f() provided by the interface of class B. for the purpose of translation, it can be viewed as a variation of a method invocation on super. The only difference is that the class whose method table should be used for dispatching is already given. Translating the above expression is essentially the same as the method invocation on super where the parent class is B.

6 Hierarchy Partitioning

This section describes why and how a class hierarchy is partitioned. As shown in the example in section 4, method tables based on Hierarchy Partitioning significantly saves on the use of run time memory. This saving is due to the sharing of method entries in each component table. Another advantage is the saving of execution time by significantly reducing the need for recomputing the method table base address. Performing adjustment of the base address of the method table during assignment reduces overhead of adding an offset at each method dispatch. In addition, few assignments will require an adjustment of the base address since adjustment is needed only if the classes of the source and the target variables are members of different partitions.

6.1 Characteristics of Partitioning Algorithm

Partitioning a class hierarchy is done in such a way that each of the resulting partitions is a single inheritance hierarchy. Observe that in each partition, two methods have the same

name if and only if one is a redefinition of the other. This observation enabled us to use the common offset technique of single inheritance for the method table of each partition. In each partition, any two methods with the same name are assigned the same index. In other words, in the component table of each partition, methods with the same name share the same entry. The final method table is constructed as a composition of the method tables of the single inheritance hierarchies.

While several partitioning algorithms are possible which satisfy the above requirement, we want the algorithm to have another desired property. The partitioning algorithm should allow a new method table to be built incrementally from the method tables of its superclasses.

A partitioning algorithm which meets these criteria is a variation of a depth first search traversal of a directed acyclic graph. In order to apply the algorithm, we first need to create a graph for the given class hierarchy. We call the graph an *inheritance graph*. In the inheritance graph, a node labeled by a name exists if and only if a class with the name exists in the class hierarchy. An arc from node X to node Y exists in the graph if and only if Y is an immediate superclass of X in the hierarchy. Obviously, an inheritance graph is a directed acyclic graph.

6.2 Traversal Algorithm

Partitioning

The algorithm for hierarchy partitioning performs a depth first traversal in an inheritance graph, visiting adjacent nodes following every arc starting from the leftmost arc. A node without any outgoing arcs (i.e. classes with no superclasses) are called a *marker*. In the traversal, each marker node represents the end of a partition. Hence, all nodes which are visited after the last marker node belong to the partition of the next visited marker node. We will call this algorithm the *traversal algorithm.*

Figure 9 shows the inheritance graph for the hierarchy shown in Figure 4 and a traversal on the graph using the traversal algorithm. A number beside a node denotes the order of visit to the node. Gray nodes in the graph denote markers. When we apply this algorithm to the class hierarchy shown in Figure 4, we get the partitioning shown in Figure 6. The numbers of partitions have been assigned so that a partition visited earlier has a smaller number. Hence the order of visits is 1, 2, 3, and 4.

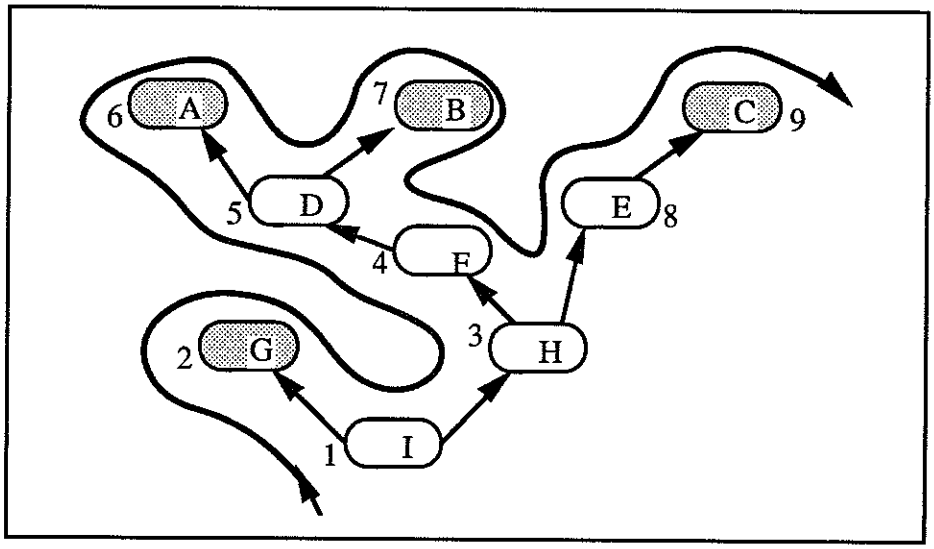

Figure 9 - Inheritance Graph and Traversal

Concatenation

A method table is created as a combination of the component tables for the partitions identified in the partitioning step. Continuing with the above example, we first need to create a method table for each partition as if the partitions were disconnected. Figure 7.A shows the method table for each partition. The final method table of class I is a concatenation of these tables. The tables are concatenated in the reverse order of visit, in order to keep the distance between two component tables within the method table of a class the same in the method tables of its subclasses, thereby allowing the common offset technique to be used for method dispatching. In the example, table 4 comes first and then 3, 2, and 1 from left to right.

6.3 Incremental Partition Algorithm

In fact, we do not need to use the traversal algorithm explicitly to create the method table of each class. Supposing that the method table of class H has already been constructed, consider the method table of class I. The method table of class H is shown in Figure 10. Observe that the method table of class I in Figure 7 contains the method table of class H. In general, the method table of a new class can be built on those of its immediate superclasses, incrementally. We call this algorithm the *incremental algorithm.*

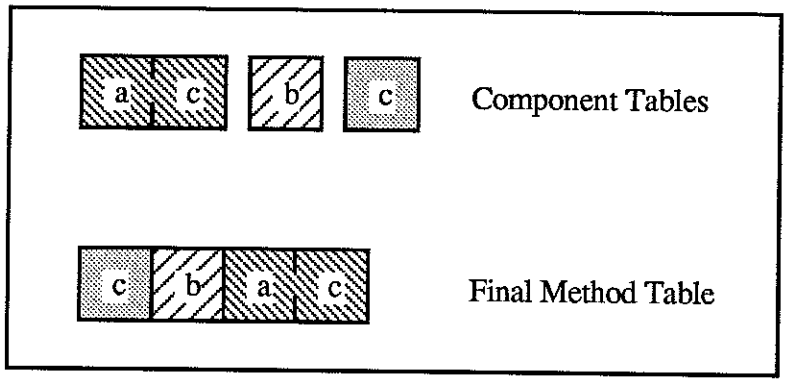

Figure 10 - Method Table of Class H

Since a class can belong to no more than one partition[1], the additional work in constructing a method table of a new class is to identify the superclass to whose partition the new class will belong. Once such a superclass is chosen, the method table of the new class can be constructed as a concatenation of the method tables of its other immediate superclasses and a component table combining the method table of the chosen super class and the methods defined by the new class. The choice of such a superclass is always unique with the graph algorithm: the leftmost child in the inheritance graph. Hence, little additional work is needed to create the method table of a new class.

We now introduce the theorem which was used in Section 5. Let P be one of the partitions created in hierarchy partitioning for class X. We say that P is a *partition for* X. Furthermore, if X is a member of P then we say P is the *partition of* X. Note that the partitions created by the traversal algorithm and the incremental algorithm are identical.

Theorem
Two classes A and B are members of the same partition for class X if and only if they are members of the same partition for any subclass of X.

The proof of this theorem is obvious since by the definition of the incremental partition algorithm, each partition for a class X will remain the same as a partition for a subclass Y, except the one which contained X. In this case, the partition of X is a subset of the partition of Y. Hence, the theorem follows.

---

[1]We assume an inheritance graph is a tree. A more general inheritance graph where a node in the graph may have more than one parent is known as repeated inheritance [Meyer 86]. An inheritance graph with repeated inheritance can be converted to a tree by replicating the class being repeatedly inherited. The technique is described in [Snyder 86]. As will be shown in later in this section, Hierarchy Partitioning works for repeated inheritance as well.

For example, consider the partitions obtained when we use Hierarchy Partitioning for class H in the hierarchy shown in Figure 4. The set of the partitions created is

{ {A,D,F,H}, {B}, {C,E} }.

Compare this with the partitions for class I, a subclass of H, as shown in Figure 6:

{ {G,I}, {A,D,F,H}, {B}, {C,E} }.

Repeated Inheritance

We now show that the hierarchy partitioning technique also works in the case of *repeated inheritance* [Meyer 88][1].

Theorem

Let k denote the index of method f in the method table of a class X. Then, f has index k in the method table for each partition containing X.

Proof of this theorem is also obvious if we consider that indices of methods in a partition do not change when a new class is added to the partition.

An example of repeated inheritance is shown in Figure 11. Class D inherits twice from A, once through B and once through C. According to the above theorem, method a will have index 0 in both B's and C's method tables as shown in the figure. In general, the hierarchy partitioning technique works for any multiple inheritance hierarchies.

One drawback of the approach is that the final method table contains duplicates in the case of repeated inheritance. For example, the method table in Figure 11 contains a twice. However, we are not aware of any other techniques which completely remove duplicates. We believe that if the common offset technique is used for repeated inheritance, duplicates can not be completely eliminated.

[1]The resulting sets are no longer partitions in a strict sense since they are not disjoint. However, the term partition will be used for this case, since it causes little confusion.

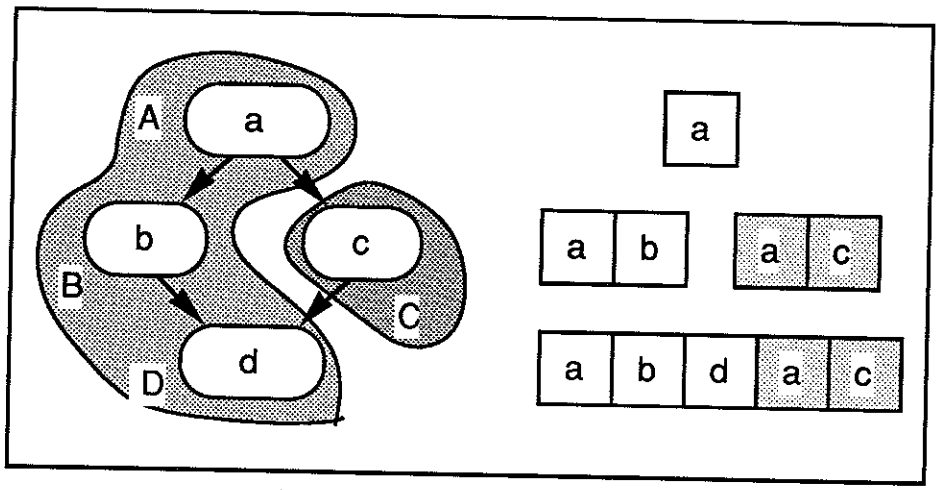

Figure 11 - Repeated Inheritance

Partitioning Strategies

When we incrementally build a method table, there are various strategies for selecting the partition of the new class. One strategy is to choose a partition which will result in the longest hierarchy chain. This strategy may have some positive effect on reducing the frequency of adjusting table base addresses. Another strategy is to choose a parent which will result in the smallest method table. A parent whose component table contains the largest number of methods which are also defined by the new class is selected. While these heuristics may be useful, there does not exist an optimal algorithm since the decision made by a class may turn out to be suboptimal when its subclasses are introduced later. Figure 12 and 13 illustrate this point. As shown in Figure 12, the partitioning for class C is optimal, since no duplicate entries exist in the method table of C. However, this decision results in a suboptimal partitioning for class D, a subclass of C, as illustrated in Figure 13.

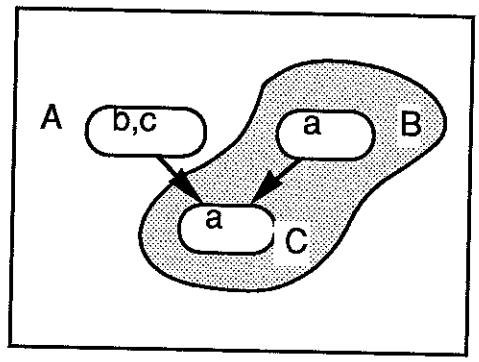

Figure 12 - Optimal Partitioning for Class C

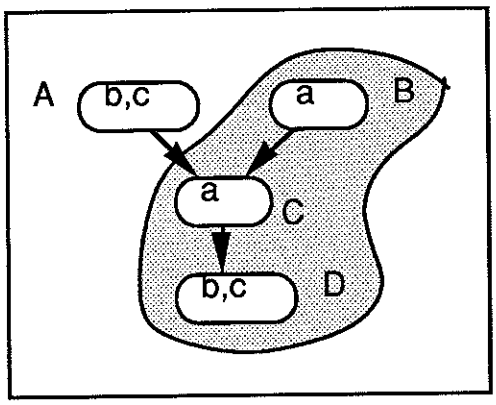

Figure 13 - Suboptimal Partitioning for Class D

7 Performance Prediction

We expect that an implementation of multiple inheritance based on hierarchy partitioning will make an efficient use of space. An example which demonstrates this expectation was shown in Figure 5. The efficiency approaches that of single inheritance as the number of classes which have only single parents increases. The extreme case of this is when the class of which the method table is created is the lowest class in a single inheritance hierarchy. Note that our scheme pays no additional overhead in either space or time in such a case. We believe that in practice, the use of single inheritance is dominant even when the language being used supports multiple inheritance. The additional space overhead caused by the support of multiple inheritance will be small. In the next section, we will show that hierarchy partitioning is more space efficient than other reported techniques.

For an analysis of the execution time overhead, we will make a rough estimate of the average execution time overhead per method dispatch. We expect that the effect of additional execution time overhead caused by the operation of adjusting the base address of a method table is negligible. This expectation is based on the following observations.

1. Typically, two or more operations are applied to the same object.

2. An adjustment of the base address is needed only when the class of the object being assigned is in a different partition from the class of the target variable.

3. Assignment from a subclass variable to a superclass variable is less common than assignment between two variables of the same class. Note that only the former case can result in the overhead of an addition operation.

An example which supports the first observation is that the user of an object usually bases its next message on the current state of the receiver. For example, a user of a stack object needs to check if the stack is empty before it issues a pop request.

The second observation has already been made in Section 5 and 6. Without a rigorous analysis, we claim that the number of assignments which require the adjustment of the table base is less than half of the total number of assignments from a subtype variable to a supertype variable. In fact, even this may be a conservative assessment. Observe that many assignments between variables of classes in two different partitions are not type compatible and so assignments between them are not allowed. For example, an assignment between a variable of class D and the one of class C shown in Figure 6 is not type compatible.

The third observation is rooted in our experience in writing programs in object-oriented languages. While subtype polymorphism is a powerful tool, its use is rather limited. The main use of subtype polymorphism is to replace the case-like explicit decision construct by moving the decision point into the receiver object. In order to put this issue in perspective, consider the frequency of case statements and assignments in programs written in a language like Pascal which does not support subtyping.

Other observations may also affect the run time overhead favorably. In contrast to conventional languages which achieve side effects with assignments, object-oriented programming achieves side effects by object encapsulation. A method invocation usually results in change in the state of the receiver object. When an assignment is used, often the target variable in the assignment is of a primitive type such as integer. Most assignments are replaced by passing an object reference in a method invocation. While argument passing involves a similar issue to assignment, parameter subtyping is not used frequently. For example, consider that C++ does not even support parameter subtyping.

Further, we hypothesize that the use of single inheritance will be dominant even when multiple inheritance is supported. Under certain assumptions, we can predict that the

increased run time overhead will be within 10 percent of the time for a method dispatch in a fast single inheritance implementation.

7 Comparison to Related Work

In this section, we compare our technique with other dispatching techniques found in the literature. Most earlier schemes proposed for method dispatching in object-oriented languages were intended for run time type checking languages. These techniques typically perform an unbounded method search at run time and so suffer from a significant overhead in execution time. More recently, Rose [Rose 88] described table based dispatching techniques for run time type checking languages. Without static type checking, method dispatching is slow. Since hierarchy partitioning is intended for a statically typed language, we will consider only those dispatching techniques proposed for statically typed object-oriented languages with multiple inheritance. The criteria used in the comparison include overheads in run time memory use, execution time, compile time, and the support of incremental compilation. Three recent proposals are reviewed in these lights.

Color indexed code technique

Dixon et al. [Dixon et al. 89] proposed an indexing scheme called the *color indexed code* technique. As in our scheme, a method dispatch is accomplished by indexing into a dispatch table at run time. A method f has the same index in every dispatch table. In order to reduce the table size, a global conflict analysis of selectors (methods) is performed using a graph coloring technique at compile time.

In comparison to our technique, the following observations can be made:

1. The color indexed technique is significantly inefficient in memory use compared with hierarchy partitioning. Each dispatch table contains many unused entries in order to place each method entry at a fixed position in every table. For example, the classes given in Figure 1 of [Dixon et al. 89] will need total 28 table entries when their coloring technique is used, while our technique will use only 15 entries.

2. Another difference between the coloring technique and hierarchy partitioning is that the coloring algorithm cannot exploit the structure of the given class hierarchies. For example, the coloring technique cannot have the efficiency of the efficient single

inheritance scheme when most classes form a single inheritance hierarchy while the efficiency of Hierarchy Partitioning approaches that of single inheritance implementation. An example illustrating this point is given in Figure 14. While our technique uses only 6 table entries, which is the minimum for the given example, the coloring technique will require 3 different colors for each method table giving a total of 9 table entries.

3. With the coloring technique, incremental compilation becomes difficult because the coloring technique depends on a global analysis of the program. Adding a new class may result in the redefinitions of dispatch tables of existing classes. This is a deficiency especially in object-oriented programming where incremental development is more pronounced.

4. The global analysis required by the coloring technique seems to incur a significant overhead in compile time. A global analysis needs to be performed every time changes occur in the set of selector names.

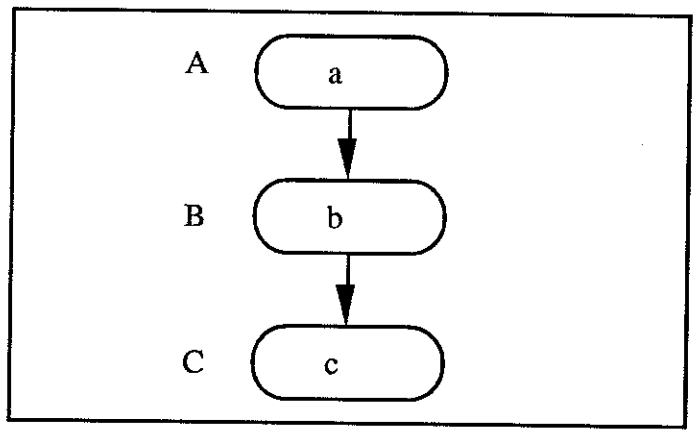

Figure 14 - An Example Class Hierarchy

Address Map Technique

Another method dispatching technique based on an *address map* was proposed by Connor et al. [Connor et al. 89]. An address map is a run time table which contains a list of offsets for methods in a method table. At run time, method dispatching involves an indirection through an address map. Although the technique was proposed for multiple inheritance, it only considers subtyping. No actual inheritance in the sense that a subclass inherits data and operations from a superclass is assumed. The distinction between subtyping and

inheritance is described in [Lee and Kafura 90B]. While the technique can also be used for run-time type-checking languages, the technique is weak in exploiting the extra information provided by compile-time type-checking. The following observations can be made in comparison to Hierarchy Partitioning.

1. While the address map technique may be suitable for the implementation of subtyping with no inheritance, there is significant overhead in execution time. An indirection taken at each method dispatch slows down the binding process.

2. An address map must be created at run time. This is an expensive process especially because there is little structural relation between subtype objects and supertype objects. Hence, an assignment between a subtype variable and a supertype variable requires an address map to be dynamically allocated and its entries to be filled at run time.

3. Each address map is an overhead in space. An address map must be created for each pair of a subtype and supertype if an assignment occurs between variables of the two types.

4. The technique does not support code sharing.

Virtual Table Approach of C++

A closely related but independently developed late binding technique used by multiple inheritance C++ is described in [Stroustrup 87B, Stroustrup 89]. As with our scheme, the technique uses a run-time table called the *virtual table* (vtbl) for each class and performs late binding by indexing into a virtual table. It seems that the virtual table approach of C++ is the best of the late binding techniques currently available for statically typed multiple inheritance object-oriented languages.

An instance of a class is a concatenation of the components called subobjects, each of which is a data structure defined by one of its superclasses or by the class itself. Each subobject contains a pointer to a virtual table. As with our method table, a virtual table contains the pointers to the virtual functions supported by the the class of the subobject. An object variable is implemented as a pointer to a subobject.

A set of example C++ classes used in [Stroustrup 89] are shown in Figure 15. Figure 16 shows an instance of class C and its associated virtual tables. As shown in the figure, each

entry in a virtual table contains two fields, namely, a method pointer and an offset of the subobject to which the method is applied. Which subobject a variable points to is determined by the type of the variable. As with our scheme, an assignment between a supertype variable and a subtype variable needs an adjustment of a constant offset to the pointer value stored in the source variable. Method dispatching at run-time involves finding the starting location of a virtual table and indexing into the table.

```
class A { virtual void f(); };
class B { virtual void f(); virtual void g(); }
class C : A, B { void f(); };
```

Figure 15 - Example Class Definitions in C++

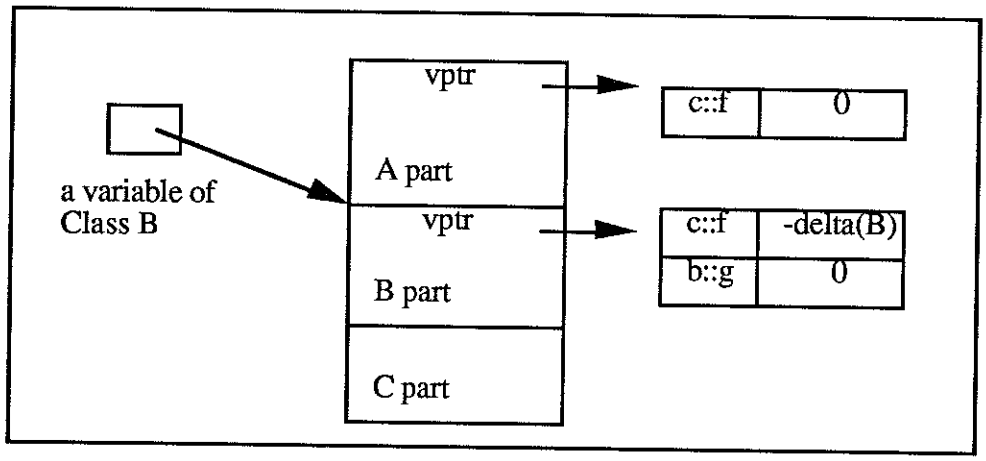

Figure 16 - An Instance of Class C

In comparison to Hierarchy Partitioning, virtual table uses the same amount of space for storing method pointers. However, the virtual table requires an additional field in each entry in order to store the offset of a subobject. Further, C++ needs to adjust the base address at each method dispatching.

A more direct comparison with the virtual table approach is possible when the issue of binding instance variables is considered. In fact, Hierarchical Partitioning is an extension of the virtual table scheme used by multiple inheritance C++. It can be shown that hierarchy partitioning is a significant improvement over the virtual table approach of C++ in both space and execution time [Lee and Kafura 90A]. Figure 17 shows the representation of the same object of Figure 16 when Hierarchy Partitioning is applied in addition to the virtual

table approach. The field for offset (delta) has been eliminated from the method table. As a result, the space used for a method table is only a half of that of a C++ virtual table. In addition, adjusting the object pointer at each invocation is not necessary.

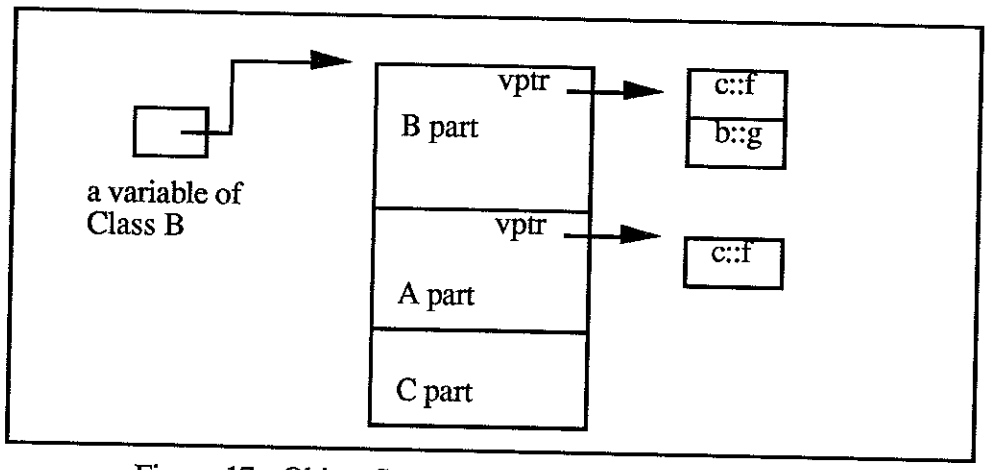

Figure 17 - Object Structure with Hierarchy Partitioning

9 Conclusion

We have described a fast and efficient dynamic method binding mechanism for statically typed multiple inheritance object-oriented languages. A technique called *hierarchy partitioning* was described in detail. Hierarchy Partitioning is an extension of the virtual function table approach and performs with an efficiency close to that of single inheritance C++ in both time and space. One of the novel features of our technique is that it exploits the structure of the given inheritance hierarchies. When many classes have only single parents, which we believe is often the case, the efficiency of our technique becomes approximately equal to that of single inheritance C++. Another contribution is that hierarchy partitioning supports incremental compilation. The technique causes little additional overhead in compile time. We provided a detailed comparison of our technique with other related techniques. We showed that our scheme is superior to those techniques. Our future plan is to use the technique in building our own language ACT++ [Kafura and Lee 89A, 89B].

10 Acknowledgement

The authors thank Mat Davis for his comments on this paper.

11 References

[Connor et al. 89] R.C.H. Connor, A. Dearle, R. Morrison, and A.L. Brown, An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance, OOPSLA '89 Proceedings, 1989.

[Dixon et al. 89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan, A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance, OOPSLA '89 Conference Proceedings, 1989.

[Goldberg and Robson 83] Adele Goldberg and David Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

[Kafura and Lee 89A] Dennis Kafura and Keung Hae Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, Proceedings of ECOOP '89 - European Conference on Object-Oriented Programming, Cambridge University Press, July 1989.

[Kafura and Lee 89B] Dennis Kafura and Keung Hae Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, Computer Journal, August 1989.

[Kafura and Lee 89C] Dennis Kafura and Keung Hae Lee, ACT++: Building a Concurrent C++ with Actors, Journal of Object-Oriented Programming, To appear in May 1990.

[Lee and Kafura 90A] Keung Hae Lee and Dennis Kafura, Hierarchy Partitioning: An Efficient Late Binding for Statically Typed Multiple Inheritance Object-Oriented Languages, In preparation.

[Lee and Kafura 90B] Keung Hae Lee and Dennis Kafura, HANA: A Model of Type and Inheritance for Object-Oriented Programming Languages, In preparation.

[Meyer 88] Bertrand Meyer, Object-Oriented Software Construction, Prentice-Hall, 1988.

[Rose 89] John R. Rose, Fast Dispatch Mechanism for Stock Hardware, OOPSLA '89 Conference Proceedings, 1988.

[Shaffert et al. 86] Craig Shaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilport, An Introduction to Trellis/Owl, OOPSLA '86 Conference Proceedings, 1986.

[Snyder 86] Alan Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, OOPSLA '86 Conference Proceedings, 1989.

[Stroustrup 86] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley 1986.

[Stroustrup 87A] Bjarne Stroustrup, What is Object-Oriented Programming?, Proceedings of ECOOP '87 - European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol 276, Springer Verlag, June 1987.

[Stroustrup 87B] Bjarne Stroustrup, Multiple Inheritance for C++, European UNIX Systems User's Group Conference, Helsinki, May 1987.

[Stroustrup 89] Bjarne Stroustrup, Multiple Inheritance for C++, An updated version of [Stroustrup 87B], AT&T Bell Laboratories, TR-??, 1989.