Space- and Time-Efficient Object Layout for Multiple Inheritance
(Extended Abstract)

Joseph (Yossi) Gil*
Department of Computer Science
Technion—IIT, Haifa 32000
Israel
yogi@cs.technion.ac.il

Peter F. Sweeney†
IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights NY 10532
USA
pfs@watson.ibm.com

Please do not distribute

Abstract

Traditional implementations of multiple inheritance bring about not only an overhead in terms of run time but also a significant increase in object space. For example, in a hierarchy of $n$ classes, the number of compiler generated fields in certain objects can be as large as quadratic in $n$. The problem of efficient object layout is compounded by the need to support two different semantics of multiple inheritance: shared, in which a base class inherited along distinct paths occurs only once in the derived class, and repeated, in which this base has multiple distinct occurrences in the derived. In this theoretical and foundational paper, we introduce a toolbox of techniques of optimization traditional object layout. The main ideas behind these techniques are the inlining of virtual bases and bidirectional object layout. Our techniques never increase the time overhead, and usually even decrease it. We show that in some example hierarchies, more than ten-fold reduction in the space overhead can be achieved. We analyze the complexity of the algorithms to apply these techniques, and give theorems to estimate the efficacy of this application. For concreteness, techniques and examples are discussed in the context of C++.

1 Introduction

Most contemporary object oriented programming languages, including Java [1], support multiple-inheritance in one form or another. This phenomena should be interpreted as an indication that the historical debate (see e.g., [5, 16]) between the proponents of the single-inheritance approach to object-oriented-programming and those in favor of the multiple inheritance one has ended with the triumph of the latter camp. It is not however clear whether this victory has remained in the methodological battle-field. Programmers now are inclined to believe that multiple inheritance is a powerful modeling technique, essential for serious object-oriented programming, and indeed, most such libraries make at least an occasional use of this feature.

*Work done while the author was at the IBM T. J. Watson Research Center
†Contact author

At the same time, the strain on system resources, both in terms of time and space overhead, that this lingual construct when combined with virtual dispatch places might be prohibitive. A recent work [26], showed that in some cases of programs that made an extensive use of multiple inheritance, as much as half of the memory space was taken by compiler generated fields which were required for realizing multiple inheritance. The time penalty of multiple inheritance is not only due to the increased number of cache misses, but also due to the complexity of data members access and to method dispatch.

This paper is concerned with the issue of reducing the overheads incurred by multiple inheritance. We make the case that by departing from conventional object models, and by applying sophisticated algorithms one can very significantly reduce the space overhead. In particular, we present a toolbox of the following five optimization techniques and transformation to an inheritance class hierarchy:

1. elimination of transitive virtual inheritance,
2. devirtualization of extraneous virtual inheritance edges,
3. inlining virtual bases,
4. bidirectional object layout, and
5. ephemeral marriage of virtual bases.

Unlike other alternative techniques, these techniques never increase the runtime cost of multiple inheritance and sometimes even decrease it.

1.1 Multiple Inheritance in C++

The C++ [25] syntax forces a programmer to select virtual and non-virtual kind (or semantics) of inheritance when the inheritance occurs. That is, the derived class must specify whether the base class is inherited is shared (virtual) if it is inherited along distinct paths in a multiple inheritance setting) or of that it is shared virtual in these circumstances). In every implementation of C++ that allows separate compilation, the kind of inheritance chosen by the programmer fixes the underlying
implementation of the derived class. This coupling between language semantics and implementation is highly undesirable. It forces the programmer to anticipate all possible contexts in which the classes may be further derived and allows only one choice for all of them. In the case of extendible libraries or any classes that have the potential to be further derived, the programmer is inclined therefore to conservatively specify the type of all occurrences of inheritance as virtual since no assumption of how the classes may be derived in the future are possible.

This predicament is even made greater by the non-negligible toll, both in terms of space and time resources, taken by the standard implementation of virtual inheritance in C++ [8]. The representation of each object of any class must include the set of offsets to all of its virtual base classes. Although these offsets can be shared among objects of the same class by storing the offsets in class tables, time-efficient implementations will repeatedly store these offsets, usually as pointers, in each instance of the class. Furthermore, these offsets or pointers are not usually shared across virtual inheritance. The time penalty is incurred when these pointers are dereferenced e.g., in an upcast, a call to an inherited (even nonvirtual) member function, or in reference to data members of the virtual base. These operations require at least one indirection; two indirections in the implementation where the offsets are stored per class and not per object.

The techniques proposed in this paper allow shared inheritance to be loosely coupled with its implementation, permitting the compiler to choose between a number of different strategies for the implementation of shared inheritance so as to minimize the space and time penalties. In other words, we break the bondage between the keyword “virtual” and the implementation, while still making sure that the implementation maintains the language semantics.

1.2 Multiple Inheritance in Other Languages

Any language that efficiently implements multiple inheritance must deal with the issues of time and space overhead. Although for concreteness, we concentrate on C++ while presenting our techniques, benefits apply to any other statically typed class-based language. This includes for example and in particular Eiffel [17], even though the semantics of multiple inheritance in it is even richer than that of C++. The impact of the techniques is even greater if all inheritance is shared (as it is with languages such as Cecil [?] and Dylan [?]).

We argue further that since these techniques belong in the back-end side of the compiler, they are applicable to other semantical models in which several implementations are to be amalgamated. An implementation of mixins in a statically typed programming such as Beta [9] constitute a perfect case in point.

1.3 Outline

The remainder of this paper is organized as follows: Section 2 gives the preliminaries to the technical discussion by describing the traditional method of laying out (C++) objects in memory. Our techniques cannot be understood nor appreciated without familiarity with this method.

The first three techniques, which share the property of requiring global program information are presented in the following Section 3. These techniques use in their implementation two graph-theoretic algorithms for the problems of transitive reduction and maximal independent set.

The two remaining techniques, which can be implemented even in separate compilation environment are presented next in Section 4. These two techniques are based on the assumption that root classes are assigned “directionality” at random, or equivalently through the application of two-wise independent hash function. This section also presents two theorems that estimate the expectation of the saving in compiler generated field. The linearity of expectation makes these estimate applicable to a wide range of metrics and benchmarks. To our knowledge, this is the first time probabilistic algorithms are used in the domain of compiler optimizations.

Section 5 demonstrates that as a whole the toolbox enables a very significant reduction in using three canonical examples. The section also justifies the use of canonical examples by arguing that in some cases saving is at least additive. In other words, we can apply optimizations to separate portions of the inheritance hierarchy, and the resulting saving is at least the sum of the saving of the portions.

In Section 6 we put our work in context by comparing it to related work.

The penultimate Section 7 points out some of the ways in which further layers of benchmarking, measurements, and heuristic should be placed upon this work. In particular the section mentions some of the algorithmic research questions involved in a global optimization in using the techniques and in combining them. This section also discusses some of the finer issues of benchmarking and measurements.

Finally, Section 8 gives the perspectives to this work and discusses our results.

Two appendices are included in the submission as background material for the reviewers. These appendices will not be included in the conference version if the paper is accepted. Appendix A presents a set of theorems that equate theoretical properties of an inheritance hierarchy with the in object compiler-generated fields that are required to support the object-oriented language features of virtual dispatch and multiple inheritance. Appendix B goes into further detail in explaining how these techniques can be used in the families optimization algorithms and presents a possible master-algorithm for their implementation.

**Terminology and notation** The nouns “instance” and “object”, are used interchangeably, as are the verbs “inherit” and “derive”.

---

3 It should be mentioned that the ability to extend observations from one language object model to that of another is not new. Myers in his OOPSLA 1995 paper [18] showed how the technique he developed for the Theta programming language can be applied to C++.
Since the implementation of virtual inheritance in the traditional layout scheme is the same, regardless of whether it is singular or multiple, we will sometimes use the term multiple inheritance in a loose sense, to include also single virtual inheritance.

Lower case letters from the beginning and the end of the Latin alphabet: \(a, b, \ldots\) and \(u, v, w, x, y, z\) denote classes. In addition, \(u_1, v_1, u_2, x_1, y_2, z\) are also used for denoting variables ranging over the domain of all classes, principally in procedures and theorems. By writing \(x \leq y\) we mean that either \(x = y\) or \(x\) inherits, directly or indirectly from \(y\). We say that \(x\) is a descendant of \(y\) and that \(y\) is an ancestor or a base of \(x\). The strict inequality \(x < y\) is used to say that \(x \leq y\) and \(x \neq y\); or in words: \(x\) is a proper descendant of \(y\) and \(y\) is a proper ancestor of \(x\).

Immediate inheritance is denoted by \(<\). Thus, \(x < y\) means that \(y\) is an immediate base of \(x\), without specifying the kind of inheritance between \(x\) and \(y\). To state that \(y\) is an immediate virtual (shared) base of \(x\) we write \(x \prec y\), whereas \(x \prec y\) means that \(y\) is an immediate nonvirtual (repeated) base of \(x\); other subscripts will be introduced as needed.

We assume that a class cannot be an immediate base of another class more than once. This assumption makes it possible to model the inheritance hierarchy of an object oriented program as a graph, rather than a multi-graph. In such a graph, which is directed and acyclic, classes are represented as nodes and immediate inheritance is represented as edges. The relationship \(x \prec y\) is represented by the edge \((x \prec y)\) leading from the node \(x\) to the node \(y\).

Finally, the reader is advised to consult Figure 1 that gives a summary of the various graphical notations used in the different kinds of diagrams used in this paper, which will include inheritance hierarchies, object layout charts, and subobject graphs.

### 2 Traditional Object Layout

Although there are many variations to it, there is basically one common scheme for laying out C++ objects in memory. The scheme, which we may call the traditional object layout scheme, is used by the vast majority of C++ compilers. Other languages that want to efficiently support multiple inheritance needs a similar layout scheme.

This section only briefly reviews the traditional scheme, for the purpose of setting out the context in which our optimization techniques take place. A detailed description of the traditional scheme can be found in standard textbooks such as [15, 24, 8]. A recent work [26], in which one of us was involved, goes into great length to compare the relative merits of the variants of this scheme in terms of the space overhead they impose.

With respect to implementing multiple inheritance there are two language features that incur a space (and time) overhead.

1. **Virtual functions** are implemented using pointers to virtual function tables discussed in Section 2.1.

2. **Virtual inheritance** is implemented using pointers to virtual bases discussed in Section 2.2.

We will see that even though the traditional approach allows some reduction in the overhead of language feature information by sharing between subobjects with repeated inheritance, the overhead can still be quite high.

#### 2.1 Pointers to Virtual Function Tables

In essence, the traditional scheme prescribes that data members are laid out "unidirectionally" in an ascending order in memory, so that the data members of each class are laid out consecutively. Also, each object or subobject belonging to a class with virtual functions has a pointer, called the Vptr, which points to the Vtbl of this class, i.e., its virtual function table. Let us first discuss nonvirtual inheritance. The layout of a base class precedes that of a class derived from it. The Vptr is commonly laid out at offset zero, which makes it possible for the Vptr of an object to be shared between subobjects with repeated inheritance, so there is in total only one VPTR in the case of single inheritance.

Several VPTRs occur in the case of multiple inheritance, since an object can share a VPTR with only one of its subobjects. Consider for example the inheritance hierarchy depicted in Figure 2.

In this hierarchy, class \(e\) inherits from both \(c\) and \(d\). Accordingly, the traditional layout of objects of class \(e\) has two VPTRs, as illustrated by the object layout chart in Figure 3.

Examining Figure 3 we see that the subobject of class \(d\) physically encompasses that of class \(b\), which in turn encompasses one subobject of class \(a\). All these three subobjects share one VPTR. Similar sharing occurs between the subobject of class \(c\) and the other subobject of class \(a\). There are two subobjects of class \(a\) since the inheritance links in Figure 2 are nonvirtual. Finally, an object of class \(e\) does not require its own VPTR(), but shares its VPTR() with that of subobjects \(d, b,\) and \(a\).
Taking a slightly wider perspective than that of C++, and adopting Eiffel [17] terminology let us call this repeated inheritance. In the current example, we may say that the class $a$ is repeatedly inherited by class $e$. A better visual illustration of this fact is given in Figure 4, the subobject graph (first introduced in [21]) of class $e$. This graph captures the containment relationships between subobjects. Evidently, the class $a$ is drawn twice in this figure.

### 2.2 Pointers to Virtual Bases

The traditional scheme ensures that in repeated inheritance the offset of a subobject $x$ is fixed with respect to any other encompassing subobject $y$ irrespective of the context of $y$, i.e., the class of the object in which $y$ itself occurs as a subobject. This is no longer true in the case of non-repeated inheritance, also known as shared inheritance, which is realized in C++ by using what is called virtual inheritance. The offset of a subobject of a virtual base class is context dependent. In order to locate such a subobject, be it for the purpose of data members access or an upcast, there is a virtual base pointer (or offset), henceforth known as VBPTR, stored in each object pointing to the subobject of the virtual base class. Consider for example the inheritance hierarchy of Figure 5, which is the same as that of Figure 2, except that $b$ and $c$ are virtually derived from $a$. In this case, class $e$ has only one subobject of class $a$.

The subobject graph of class $e$ in this case is given in Figure 6. This graph makes it clear that there is only one subobject of class $a$, which is shared between the subobjects of classes $b$ and $c$.

Even though virtual inheritance is a lingual mechanism designed to support a shared variant of multiple inheritance, the
C++ semantics allows also single virtual inheritance. Thus, the fact that the in-degree of a class is greater than one in a subobject graph is a necessary but an insufficient condition that the class is a virtual base. This is the reason behind the notational convention of drawing a circle around names of virtual bases, as was the case with class \( a \) in Figure 6.

The memory layout of objects of class \( e \) is given in Figure 7, which shows how VBPTRs are used to realize the sharing of a VBPTR between subobjects of classes \( b \) and \( d \).

Examsing the figure, we can also see that since objects of class \( d \) occupy a contiguous memory space, it must be the case that the offset of the subobject of class \( a \) with respect to the data members of \( d \) is different in objects of class \( d \) than in objects of class \( e \). Resuming our VPTRS counting, we see that objects of class \( e \) have in total three VPTRS: two for the immediate parents of \( e, c \) and \( d \), and one for the subobject of the virtual base \( a \). The VPTR of \( d \) is also shared with \( e \) and \( b \). In contrast, the VPTR of \( a \) cannot be shared with any of its descendants, since its relative offset with respect to these is not fixed.

As explained above, the offsets to virtual base classes must be stored in memory. In the variant described above these offsets are stored as VBPTRS in each instance of the class. A time penalty is incurred when these pointers are dereferenced for e.g., an upcast, a call to an inherited (even nonvirtual) member function, or in accessing a data member of the virtual base.

Alternatively, to reduce the space overhead, virtual base offsets may be stored in class tables, frequently as special entries in the VTBLS. This variant, although more space efficient in the case of many objects instantiated from the same class, doubles the time penalty since each access to members of the virtual base must pass through two levels of indirection instead of one. We argue that the potentially significant reduction in the number of VBPTRS provided by our techniques, makes the savings of this variant less attractive.

It turns out that for any given class, the number of VBPTRS stored in each object in one variant is exactly the same as the number of offsets stored in the class information in the other variant. For the sake of concreteness, we can therefore concentrate in the “time-efficient” variant in which pointers to virtual bases are stored in objects.

The number of VBPTRS is greater than what it might appear at first since these pointers can not be shared across virtual inheritance. To see why, let’s look at the hierarchy of Figure 8. Each instance of class \( u_1 \) has a virtual base pointer to the \( v_1 \) subobject. This is also the case for instances of class \( v_2 \). Now, since the inheritance link between \( v_2 \) and \( u_1 \) is nonvirtual, then the VBPTR to \( v_2 \) can be shared by \( u_1 \) and \( u_2 \). Also, each instance of class \( u_2 \) must store two pointers to both the \( v_1 \) and the \( v_2 \) subobjects which correspond to virtual bases. However, as depicted in Figure 9, the pointer to the \( v_1 \) base is duplicated in a \( u_2 \) instance: there is one such pointer in the memory area allocated for \( u_2 \)’s own data, but also another such pointer stored in the \( v_2 \) subobject of \( u_2 \).

Let us make the distinction between essential and inessential VBPTRS. The essential VBPTRS are precisely the minimal set

\[ \{ u_1, u_2 \} \]

In its C++ compilers, Microsoft uses a variant of this approach. VBPTRS are in VTBLS that are separate from the VTBLS. This approach may require additional VTBLS pointers in an object.
of VBPTRs which allows direct or indirect access to every virtual subobject from any of its containing subobjects. Inessential VBPTRs are those which can be computed from the essential ones, but are stored to ensure that an upcast to an indirect virtual base takes no more time than an upcast to a direct virtual base, thus guaranteeing constant access to all data members and all virtual functions. More generally, in the traditional model, there is no sharing across virtual inheritance links of any compiler-generated field, including VPTRs and other fields used for realizing run-time type information. Inessential VBPTRs are introduced whenever essential VBPTRs are not to be shared across virtual inheritance links.

Alternatively, to reduce space overhead in objects, inessential VBPTRs could be eliminated. This translates, in our example, to having only one VPTR to \( v_2 \) that would be stored in the \( v_2 \) subobject of \( u_2 \). This more space efficient variant increases the time to access a virtual base subobject when a chain of VBPTRs has to be followed. In our example, if inessential VBPTRs are eliminated, accessing the \( v_2 \) subobject from the \( u_2 \) object requires two levels of indirection instead of one.

More generally, each instance of the bottom most class in a virtual inheritance chain of \( \eta \) classes, as shown in Figure 10, must include \( \eta(\eta-1)/2 \) pointers in total. The situation is no different if virtual bases are stored with class information, except that the overhead is not repeated per object. The number of offsets that must be stored in total for all classes is \( (\eta^2 - \eta)/2 \), i.e., cubic in the number of classes in the hierarchy!

This paper introduces techniques that allow significant space reduction without incurring time overhead.

3 Streamlining Virtual Inheritance

Streamlining virtual inheritance consists of three class hierarchy transformations. The first is the elimination of transitive virtual inheritance edges. This transformation brings the hierarchy into a more canonical form, making it easier to apply the two subsequent transformations which are space optimization techniques.

Next we devirtualize those virtual inheritance edges which are not used for designating sharing. Finally, we identify some of the cases where virtual inheritance can be inlined. We present two versions of the inlining algorithm. Of these, the more powerful (and the computationally more expensive) one subsumes devirtualization, and in fact eliminates the need for a separate edge devirtualization transformation. Thus, the discussion of devirtualization is predominantly for expositional purposes.

Although each transformation can be applied to an inheritance hierarchy on its own, the order we chose is the one which maximize their combined benefit.

The first two transformations have been discussed separately in the literature. We bring these two into synergy here with the new inlining technique, while correcting and generalizing a previously published algorithm for devirtualization.

The complexity analysis of all the procedures in this section are in terms of the class hierarchy graph.

3.1 Transitive Virtual Inheritance Edges

We now present the process of eliminating transitive virtual inheritance edges. This transformation simplifies the class hierarchy, whereby making it easier to reason about it. This transformation should be applied prior to any optimization techniques.

Suppose that class \( y \prec_0 x \) and \( z \prec_0 y \). Then, the transitive virtual inheritance edge \( \langle z \prec_0 x \rangle \) overspecifies that \( x \) is a virtual base of \( z \). In laying out \( z \), it is immaterial whether \( x \) is a direct or an indirect virtual base of \( z \), and therefore we can eliminate the transitive edge \( \langle z \prec_0 x \rangle \). This phenomena was first observed by Tip and Sweeney [27] who also showed how to remove transitive virtual inheritance edges. Procedure eliminate-
transitive-virtual-edges of Figure 11 shows how this is done.

Let \( n \) be the total number of classes in the hierarchy and let \( m \) be the total number of inheritance edges. There is empirical evidence indicating that in practice \( m \) is linear in \( n \); Krall et al. [13, Table 2], examining seven application that were written in four different languages and had the number of classes varies from 225 to 1,802, found that \( m \leq 1.89n \).

It is possible to compute the \( \leq \) relationship between any two classes in the hierarchy in time \( O(nm) \), by e.g., doing a breadth-first search from each class. In practice, this approach will be more efficient than the naïve transitive closure algorithm which runs in \( O(n^2) \) time. Once computed, the \( \leq \) relationships can be stored using in \( O(n^2) \) space. As the number of classes in a class hierarchy is in the order of couple of thousands, such runtime bound is not prohibitive. We henceforth assume that such preprocessing has been applied.

With this preprocessing, eliminate-transitive-edges time is \( O(n^2) \) in a naive implementation. A better implementation which initiates a depth-first search from each node \( x \), looking for backward edges will again result in \( O(nm) \) computation time.

Consider the class hierarchy of Figure 12(a). Then, the application of eliminate-transitive-edges will result in the hierarchy of Figure 12(b), where the following edges have been removed: \( (d \prec_\circ c), (d \prec_\circ b), (e \prec_\circ b) \) and \( (g \prec_\circ b) \).

Clearly, global program information is a prerequisite of eliminate-transitive-edges.

It should be stressed that eliminate-transitive-edges is merely a graph transformation technique, and not a C++ semantic preserving source to source transformation. There are rather subtle semantic differences at the source level between the Figure 12(a) and Figure 12(b). For example if a class \( z \) virtually inherits from a base class along two paths, one of which is protected and the other is private, then eliminating the protected virtual inheritance path will change the semantics of \( z \). Therefore, this transformation needs to be done after static semantic checking. Conversely, the same example demonstrates a rationale of a program using transitive inheritance edges.

3.2 Edge Devirtualization

Procedure eliminate-transitive-edges removes redundant virtual edges. The next step in the streamlining of virtual inheritance is to devirtualize those virtual edges in a hierarchy which do not represent shared inheritance. This semantic preserving transformation is our first space optimization technique. Devirtualizing an edge allows VBPTRs to be eliminated, and it opens opportunities for sharing compiler-generated fields.

As a simple example, consider a hierarchy with two classes, \( x \) and \( y \), where \( y \) virtually inherits from \( x \). Then, the edge \( (y \prec_\circ x) \) can be devirtualized by replacing it with the edge \( (y \prec_\circ x) \). Devirtualization was first proposed in [2], however, determining when it is legitimate is quite an illusive prospect. For example, \( (y \prec_\circ x) \) must not be devirtualized if two more classes were to be added to our little example, to form the hierarchy of Figure 13. The reason is that there are two subobjects of type \( y \) in a \( x \) object, and devirtualizing \( (y \prec_\circ x) \) would also imply two \( z \) subobjects in \( z \), which violates virtual inheritance semantics. Indeed, this is a case where the devirtualization algorithm of [2] fails.

Thus, in contrast to prior belief, a virtual base with a single incoming edge cannot be devirtualized without a global exami-
[1] **Function** is-duplicated(Node y): Boolean
[2] Begin
[3] For each u \subset y do
[4] if is-duplicated(u) then
[5] Return true
[6] fi
[7] For each v \subset y, v \neq u do
[8] if HCD(u,v) then
[9] Return true
[10] fi
[12] od
[13] Return false
[14] end

Figure 14: A function that determines if a class is duplicated.

The complexity of is-duplicated and HCD can be greatly improved by using a standard memoization technique. The total run-time of a memoized version of HCD in \( t \) invocations is \( O(n^2 + t) \). Applying a memoized version of is-duplicated to all classes in the hierarchy requires \( O(nm) \) time. To see this, note that the amortized complexity is-duplicated(y) is \( (d(y))^2 \), where \( d(y) \) is the in-degree of \( y \). Then, the total run-time of all \( n \) applications of is-duplicated is in the order of

\[
\sum_{y \in H} d(y)^2.
\]

Since \( \sum_{y \in H} d(y) \) is at most \( m \), and since \( d(y) < n \) for all \( y \), the above is maximized when there are \( m/n \) nodes for which \( d = n \). We obtain that the total run time in applying a memoized version of is-duplicated-single-VI is

\[
O(n^2 + \frac{m}{n}n^3) = O(nm).
\]

Devirtualization can still be done even if there are multiple incoming virtual edges into a virtual base \( \alpha \).

---

[1] **Function** HCD(Node v1, v2): Boolean
[2] Begin
[3] For each w \in H do
[4] if \( w \leq v1, w \leq v2 \) then
[5] Return true
[6] fi
[7] od
[8] Return false
[9] end

Figure 15: A function that determines if two classes have a common descendant.

Figure 16: An example of a multiple incoming virtual edges, some of which may be devirtualized.

**Definition 1** A class \( y \) is duplicated in a hierarchy \( H \) if there are multiple occurrences of \( y \) in the subobject graph of some class \( z \) of \( H \).

There are multiple occurrences of \( y \) in \( z \) if, for example, \( z \) repeatedly inherits from \( y \) “more than once”, or if \( z \) inherits from \( y \) in both a repeated and shared manner. Also, \( y \) is duplicated if there is yet another duplicated class \( u \) which non-virtually and directly inherits from \( y \). More precisely, the recursive boolean function is-duplicated in Figure 14 gives an algorithm for this definition. This function uses the auxiliary HCD function of Figure 15 which determines if two classes Have a Common Descendant.

The complexity of is-duplicated and HCD can be greatly improved by using a standard memoization technique. The total run-time of a memoized version of HCD in \( t \) invocations is \( O(n^2 + t) \). Applying a memoized version of is-duplicated to all classes in the hierarchy requires \( O(nm) \) time. To see this, note that the amortized complexity is-duplicated(y) is \( (d(y))^2 \), where \( d(y) \) is the in-degree of \( y \). Then, the total run time of all \( n \) applications of is-duplicated is in the order of

\[
\sum_{y \in H} d(y)^2.
\]

Since \( \sum_{y \in H} d(y) \) is at most \( m \), and since \( d(y) < n \) for all \( y \), the above is maximized when there are \( m/n \) nodes for which \( d = n \). We obtain that the total run time in applying a memoized version of is-duplicated-single-VI is

\[
O(n^2 + \frac{m}{n}n^3) = O(nm).
\]

Devirtualization can still be done even if there are multiple incoming virtual edges into a virtual base \( \alpha \).

---

For example, in Figure 16, \( \langle y \subset_0 x \rangle \) is the single virtual inheritance edge. A single virtual inheritance edge \( \langle y \subset_0 z \rangle \) can be safely devirtualized since it is not used to designate that any other class \( z \) has \( x \) as a “shared base”.

For example, class \( q \) in Figure 12(b) has two incoming virtual edges: \( \langle b \subset_0 a \rangle \) and \( \langle h \subset_0 a \rangle \). Since there are no common descendants to the nonduplicated nodes \( b \) and \( h \), both these edges represent single virtual inheritance, and can be devirtualized. Conversely, there are two incoming virtual edges \( \langle d \subset_0 c \rangle \) and \( \langle e \subset_0 c \rangle \) into class \( c \). However, since \( g \) is a common ancestor of \( d \) and \( e \), these edges are not a case of single virtual inheritance.

Procedure eliminate-single-VI of Figure 17 devirtualizes all single virtual inheritance edges. This algorithm improves on the result of [2] by considering multiple incoming virtual inheritance edges.

Using amortization analysis, it is easy to see that the run time of eliminate-single-VI is maximal when the sum of squares of in-degrees is maximal. Using the same considerations as in the analysis of is-duplicate we obtain that the run time of eliminate-single-VI is \( O(n^2m) \). When procedure eliminate-single-VI is applied to Figure 12(b) edges \( \langle b \subset_0 a \rangle , \langle h \subset_0 a \rangle \), and \( \langle f \subset_0 d \rangle \) are devirtualized resulting in the hierarchy of Figure 12(c). Consider the subobject graph in Figure 18 of class \( g \) in Figure 12(a). There
are 6 VPTRs and 10 VBPTRs. After eliminate-extraneous-VI has been applied, the number of VPTRs is reduced by one to 5 and VBPTRs is reduced by four to 6.

Recall the chain of \( n \) classes that was presented in Figure 10. Initially, an object of class \( C_n \) requires \( n \) VPTRs and a quadratic number of VBPTRs. After applying eliminate-extraneous-VI to this chain all inheritance is devirtualized, and an object of class \( C_n \) requires only one VPTR and no VBPTRs.

The size of a \( y \) object may be reduced in a number of different ways due to the devirtualizing of an edge \( \langle y \prec_C z \rangle \). First, the essential VBPTR from \( y \) to \( z \) is always eliminated. Second, the devirtualization enables sharing between \( z \) and \( y \) of compiler generated fields. These include one VPTR that maybe shared between \( y \) and \( z \). And even greater is the saving potential in the inessential VBPTRs from \( y \) to the virtual bases of \( z \), which are all eliminated. There are up to \( n \) of these.

In addition, \( y \)'s saving occur every time \( y \) is a subobject in some object \( z \). There could be an exponential number in \( n \) of \( y \) subobjects in an object \( z \). Another kind of potential saving are the inessential VBPTRs to \( z \) in the subobjects derived from \( y \). There are up to \( n \) possible class that are derived from \( y \). Each one of these classes has the potential to have exponential in \( n \) number of subobjects derived from \( y \).

### 3.3 Inlining Virtual Bases

This section introduces the last class hierarchy transformation: inlining virtual bases. By inlining we mean that instead of storing a pointer to a virtual base subobject, this subobject can be stored in a fixed offset in the memory layout of the derived class. For an example, let's go back to the subobject graph of Figure 6. Instead of laying out class \( C \) as in Figure 7, inlining \( a \) into \( b \) obtains the layout shown in Figure 20. The new layout eliminates the VBPTR from \( b \) to \( a \), and the separate VPTR for \( a \).

Inlining is similar to devirtualization in that compiler-generated fields are eliminated since the offset of a virtual base is fixed with respect to a derived class. However, unlike devirtualization, an inlined base may still be shared. The potential savings associated with inlining include those of devirtualization. Furthermore, additional inessential VBPTRs can be eliminated. Suppose, for example, that inlining is applied to the hierarchy of Figure 21. Assume that \( a_i \) is inlined into \( a_{i+1} \) and \( b_j \) is inlined into \( b_{i+1} \) for \( i = 1, \ldots, n-1 \). Clearly, a subobject of class \( a_i (b_j) \) does not need now any VBPTRs to \( a_j(\bar{b}_j), j < i \).
Two VBPtrs, one from $a_2$ to $b_1$ and the other from $b_2$ to $a_3$, are sufficient for any $a_i$ (respectively $b_j$) to access any virtual base $b_j$ (respectively $a_j$) $0 < j < i$. This is because inlining makes the offsets of all $b_j$ (respectively $a_j$) subobjects fixed with respect to each other. Therefore, the total number of VBPtrs in objects of class $c$ is reduced from $(n-1)(n-2)$ to 2, i.e., from quadratic to a constant.

As mentioned above, if $x$ is a virtual base that has an immediate duplicated descendant $y$ then $x$ must not be inlined into $y$. This is because only one virtual base subobject of $x$ occurs in the subobject graph while multiple subobjects of $y$ occur. For example, consider the class hierarchy in Figure 13. If $x$ were inlined into $y$ which is duplicated in $x$, then there would be two subobjects of type $x$ in a $x$ object, contradicting the semantics of virtual inheritance.

This section introduces two procedures that implement the inline virtual base transformation. The first procedure is based on the simple observation that a virtual base can be inlined into at least one of its immediate nonduplicated descendants. Assuming that procedures eliminate-transitive-virtual-edges and eliminate-single-VI were run, a simple algorithm for selecting a derived class in which to inline a virtual base is given in Figure 22.

Procedure simple-inline-VB introduced a new kind of inheritance edges. In writing $y \prec_{c} x$ we mean that $x$ is an immediate virtual base of $y$ and also that $x$ is inlined into $y$. An edge of this kind is drawn using double lines in our diagrams.

The exists statement in Figure 22 is nondeterministic, and it is not clear a priori which descendant to inline into. For example, in the subobject graph shown in Figure 6, $a$ could be inlined into either $b$ or $c$, but not into both. It seemed better to inline it into $b$, since this inlining reduces the size of instances of three classes ($b$, $d$ and $e$) as opposed to only two classes ($c$ and $e$) if the inlining was into $c$.

What should be the general decision rule? The answer to this recalcitrant question is left to future experimental and theoretical research. There are many different heuristics, algorithms, and experiments that should be explored. Since the discussion of these issues transcends the usual scope of a concluding section of a conference paper, we have included it in Appendix B.

We now introduce a more powerful version of inlining virtual bases, inline-VB presented in Figure 23. It is based on the observation that a virtual base may be inlined into more than one of its nonduplicated subobjects provided that they do not have a common descendant. To understand the algorithm recall that a set of nodes is independent in a graph, if no two nodes in it are connected by an edge.

The maximal independent set problem is to find an independent set that maximizes the number of its members. We could make an effort to choose one strategy which is likely to be better, by selecting $S$ in procedure inline-VB to be a maximally independent set. If this is done, then inline-VB covers edge de-virtualization; if an $y \prec_{c} x$ would have been devirtualized by eliminate-single-VI, then in the graph $G$ of inline-VB, node $y$ would have no edges incident on it, and therefore would be part of the maximal independent set.

Unfortunately, the maximal independent set problem is known to be NP complete [10]. This means that the best way, at
least to the extent known so far to human-kind, of finding such a set is not significantly better than trying out all possible different sets $S$. Although this exponential computation time sounds deterring, \textit{inline-\textit{\text{\textit{}}}} may be feasible in many cases, since it is exponential only in the number of immediate virtual descendants, which could be a small number in practice.

More precisely, the total run-time of \textit{\textit{\textit{}}\textit{\textit{}}-\textit{\textit{\textit{}}}} is in the order of

$$\sum_{x \in H} 2^{d(x)}$$

where $d(x)$ is the number of incoming virtual edges of a class $x$. We conjecture that $d(x) = O(\log n)$ in most practical hierarchies, which means that the \textit{\textit{\textit{}}\textit{\textit{}} run-time of \textit{\textit{\textit{}}\textit{\textit{}}} is polynomial. In addition, we tend to believe that applying heuristics for the maximally independent set problem may give good results. This belief has to be confirmed by an experimental study. Appendix B explores a family of algorithms that use heuristics to implement \textit{\textit{\textit{}}\textit{\textit{}}}.

### 3.4 Summary

This section discusses how the transformations presented above can be combined together into two cohesive algorithms.

For any such algorithm, \textit{\textit{\textit{}}\textit{\textit{}}} should always be applied first to a class hierarchy before either edge devirtualization or inlining. To see this consider the transitive edge $\langle a \prec \theta, d \rangle$ in Figure 12(a). \textit{\textit{\textit{}}\textit{\textit{}}} enables \textit{\textit{\textit{}}\textit{\textit{}}} since only after the transitive edge $\langle a \prec \theta, d \rangle$ is eliminated, can $\langle a \prec \theta, \bar{b} \rangle$ be devirtualized. \textit{\textit{\textit{}}} eliminates inferior inlining candidates. Inlining $a$ into $d$ attains the same benefits or less than inlining $a$ into $b$, and therefore is inferior. But $\langle a \prec \theta, \bar{d} \rangle$ is a transitive edge. It is important to execute the procedures \textit{\textit{\textit{}}} and \textit{\textit{\textit{}}} before \textit{\textit{\textit{}}} to ensure that only shared bases are inlined.

There are two natural ways to combine the above transformations. Figure 24 presents a combined algorithm whose overall execution time is bounded by the complexity of \textit{\textit{\textit{}}\textit{\textit{}}}Figure 25 presents a combined algorithm whose overall execution time is bounded by the complexity of \textit{\textit{\textit{}}} Since \textit{\textit{\textit{}}} will inline single virtual inheritance edges, \textit{\textit{\textit{}}} is not needed in Figure 25.

Both the elimination of transitive inheritance edges and edge devirtualization or \textit{\textit{\textit{}}} are needed to remove the circle notation which identifies shared bases in a class hierarchy. The application of \textit{\textit{\textit{}}} and \textit{\textit{\textit{}}} makes the circle notation for virtual classes in subobject graphs redundant. Nevertheless, we retain it because it highlights virtual bases. If these procedures have been applied, then shared bases are exactly those nodes in the subobject graph whose in-degree is greater than one.

Of the three transformations presented, only \textit{\textit{\textit{}}} is a source-to-source transformation. As noted above, \textit{\textit{\textit{}}} must be applied after static semantic checking of the program. \textit{\textit{\textit{}}} must be performed on an intermediate representation of the application as there is no analogous language construct with which to represent inlined inheritance. Therefore, either version of \textit{\textit{\textit{}}} must be applied to an application’s intermediate representation.

In summary, although \textit{\textit{\textit{}}} has the potential to dramatically reduce the number of compiler-generated fields, we believe that its benefit in practical situations might be limited since it is unlikely that a human will deliberately use single virtual inheritance in a single cohesive and isolated application. Its impact will be most noticed in hierarchies which are generated automatically [27], and in cases where class libraries used in ways which transcend the intent of their designer.

On the other hand, the decision of whether \textit{\textit{\textit{}}} should be invoked when whole program information is available (in such systems as the IBM Visual Age C++ compiler [19, 12], Vortex [7]) is not a hard one. The procedure \textit{\textit{\textit{}}} will never increase the execution-time or memory consumption of an application. Moreover, the run-time of the first two transformations and the \textit{\textit{\textit{}}} are only polynomial. The run-time of \textit{\textit{\textit{}}} depends on the extent of the optimization that is applied. There are, however, good heuristics for the maximal independent set that run in polynomial time.

### 4 Bidirectional Layout Techniques

Given the inheritance hierarchy of Figure 26, the traditional scheme will layout all the classes using only one \textit{\textit{\textit{}}} except for class $c$ which requires two \textit{\textit{\textit{}}}.

We can layout class $c$ using only one \textit{\textit{\textit{}}} as well as follows. Suppose that class $\theta_1$ is laid out using negative offsets. That is to say, its \textit{\textit{\textit{}}} will be at offset zero, and all its data members are laid out in decreasing addresses. This will force what we may call a \textit{\textit{\textit{}}} all classes $\theta_1, \ldots, \theta_9$.
Similar layout is imposed on the VTBL: functions associated with classes $a_1, \ldots, a_9$ will occupy entries $-1, -2, \ldots$ in their table. Classes $b_2, \ldots, b_9$ will still have a positive directionality, with their entries at offsets $0, 1, \ldots$ in VTBL. Classes $a_9$ and $b_5$ are married in class $c$: they share their VPTR as illustrated in Figure 28, and their VTBLs are juxtaposed as illustrated in Figure 28.

The directionality of a class $x$ is denoted by $\chi(x)$. In the traditional scheme, $\chi(x) = \text{positive}$ for all $x$. With bidirectional layout, $\chi(x)$ can be either positive or negative. If this is the case, then we say that $x$ is directed.

Two more values which $\chi(x)$ can assume are

1. **mixed**, which is used if $x$ shares its VPTR with two base classes that are married with each other, and

2. **none**, which occurs if $x$ and all of its base classes have no virtual functions and consequently $x$ has no need for a VPTR.

In both cases we will say that $x$ is undirected. The predicate

$$\chi(x) = -\chi(y)$$

means that either $\chi(x) = \text{positive}$ and $\chi(y) = \text{negative}$ or that $\chi(x) = \text{negative}$ and $\chi(y) = \text{positive}$. The semantics of the different values of $\chi(x)$ are summarized in Table 1.

In order for bidirectional layout to work in a separate compilation setting, we need that an oracle will assign the right directionality to classes $a_1$ and $b_2$ when they are compiled, which could be prior to the compilation of class $c$. When no supernatural powers are at our disposal, a simple and effective work around is to assign directionality at random. Figure 29 displays procedure $\text{assign-initial-directionality}$ which might be used for assigning directionality to classes whose directionality is not determined by their parents. This will insure that with probability $0.5$, one VPTR will be saved in class $c$. We can say that the expected saving is $0.5 \cdot 1 = 0.5$ VPTR.

The crucial point in computing this expectation is that the “coin-tosses” in $a_1$ and $b_2$ were independent. More generally, an expected saving can be guaranteed if any two selections of directionality to root classes are independent. It is not necessary however to have independence between any three selections. To implement “pair-wise-independence” random selection we can apply a standard technique [6] of randomized algorithms and replace the coin tosses by a hash function. In other words, whenever a compiler encounters a class whose directionality is not forced, it applies a hash function, selected at random from a universal class of such functions, to its name.

Table 1: Semantics of the different values of $\chi(x)$.

<table>
<thead>
<tr>
<th>$\chi(x)$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>$0, 1, \ldots$</td>
</tr>
<tr>
<td>negative</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>$-1, -2, \ldots$</td>
</tr>
<tr>
<td>mixed</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>$-2, -1, 0, 1, \ldots$</td>
</tr>
<tr>
<td>none</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>none</td>
</tr>
</tbody>
</table>

$a$: VPTR at offset zero
$b$: data members in negative offsets
$c$: data members in positive offsets
$d$: indices of VTBL entries

Figure 26: A hierarchy used to exemplify bidirectional layout.

![Figure 26: A hierarchy used to exemplify bidirectional layout.](image)

Figure 27: Bidirectional layout of class $c$ of Figure 26.

![Figure 27: Bidirectional layout of class $c$ of Figure 26.](image)

Figure 28: Bidirectional layout of the VTBL of class $c$ of Figure 26.

![Figure 28: Bidirectional layout of the VTBL of class $c$ of Figure 26.](image)

[1] **Procedure** $\text{assign-initial-directionality}(Node n)$

[2] **Begin**

[3] **If** $n$ has no virtual functions **then**

[4] $\chi(n) \leftarrow$ none

[5] **else**

[6] $\chi(n) \leftarrow \text{Random}(n)$

[7] **fi**

[8] **end**

Figure 29: A procedure for assigning initial directionality to a class.
4.1 Ephemeral Marriage of Virtual Bases

The use of indirection in the implementation of virtual bases makes it possible to place them anywhere in memory. This degree of freedom, together with bidirectional layout unfolds saving opportunities beyond those suggested by our motivating example. Let v₁ and v₂ be two virtual bases, direct or indirect, of class u, and suppose that χ(v₁) = positive and χ(v₂) = negative. Then, between v₁ and v₂ we could save one VPTR, by placing them against each other in the layout of u. We say that v₁ and v₂ are married in u, but in contrast with the marriage of nonvirtual base classes a₀ and b₅, this marriage is ephemeral. Subobjects v₁ and v₂ are not necessarily married with each other in every context in which they occur together. In other words, even though the subobjects of v₁ and v₂ are adjacent in objects of class u, they are not necessarily adjacent if u occurs as a subobject of another class w, w < u. Therefore, it is necessary that u maintain two VBPTRs, one for v₁ and one for v₂. (In certain conditions, it is also possible to save a VBPTR by using only one VBPTR to reference the combined subobject of v₁ and v₂. These conditions are explored further in Appendix B.3 below.)

Procedure ephemeral-virtual-base-marriage of Figure 31 implements this technique, and even extends it noting that a class may be temporarily married with one of its parents.

The procedure assumes that a directionality was already assigned to the class and to all of its parents. In particular, the procedure is expected to be executed after the procedure (presented below) that persistently marries nonvirtual bases. Note the run-time of this procedure on all nodes in the hierarchy is linear in the size of the hierarchy.

Consider again the hierarchy of Figure 5. Suppose that χ(a) = positive and that χ(e) = negative. Then, procedure ephemeral-virtual-base-marriage improves further the layout of Figure 20 obtaining the layout of Figure 32 which uses only one VPTR and one VBPTR. Notice, that unlike nonvirtual inheritance, the virtually derived classes of a base may have different directionals.
The marriage of two virtual bases requires that their VTBLs are juxta-posed. Since in general, a class has a different VTBL for every context this class is used, marriage incurs no additional overhead. When classes \(v_1\) and \(v_2\) are married in \(u\), we also place the VTBL of \(v_2\) class in a \(u\) context against the VTBL of \(v_2\) in a \(u\) context. If the VTBL of (say) \(v_1\) in a \(u\) context happens to be exactly same as that of a derived class of \(u\), \(w\), then the marriage of \(v_2\) may make it impossible to optimize class space by using only one VTBL for \(v_2\) in \(u\) and for \(v_2\) in \(u\).

### 4.2 Persistent Marriage of Nonvirtual Bases

Let us now proceed to the description of bidirectional layout for nonvirtual bases. Let us assume inductively that a directionality was assigned to all classes from which a class \(u\) inherits, and that all these classes were laid out already. The questions are then how should \(u\) be laid out, what kind of sharing of VPTRs will \(u\) have with its parents, and what should \(\chi(u)\) be. Procedure \texttt{bidirectional-layout} of Figure 33 answers these questions, by detailing how the nonvirtual bases of \(u\) are married together in \(u\).

The cases in procedure \texttt{bidirectional-layout} in which \(u\) has no parents, only one parent, or two parents which both are of directionality none are rather pedestrian. The case where \(\chi(v_1) = -\chi(v_2)\) is the most interesting one since it is the only case in which a VPTR is saved. Note that the procedure favors a \textit{none} directionality for \(u\). However, as in the vast majority of cases, this is not possible, it tries to make \(u\) directed, in or- der to leave open future optimization opportunities. Class \(u\) is assigned a mixed directionality only if there is no other choice.

When \(u\) has more than two parents procedure \texttt{bidirectional-layout} calls procedure \texttt{pairup} (presented in Figure 34). Procedure \texttt{pairup} simply generalizes the breakdown into different cases in \texttt{bidirectional-layout} when \(u\) has two parents. Interesting, \texttt{pairup} could be used to replace \texttt{bidirectional-layout} completely!

### 4.3 Efficacy Analysis

The following definition is pertinent for this.

**Definition 3** The spanning forest \(T\) of a subobject graph \(G\) is obtained from \(G\) by removing from it all edges incoming into its virtual nodes. Let \(n_e\) be the number of leaves in the trees of \(T\) and let \(n_L\) be the number of these leaves which correspond to distinct classes.

More generally, we can assert:

**Theorem 1** By assigning random directionality to all classes whose directionality is not dictated by their ancestors, the expected saving in the number is of VPTRs at least \(n_L/2\).

**Proof:** The proof uses some properties of binary trees, as well as the fact that in addition to simple bidirectional layout, we may also save VPTRs by marrying together virtual bases of opposite directionality. This technique is presented below in Section 4.1. The proof details are omitted.

---

Figure 33: A procedure for a bidirectional layout of a class assuming that all its parents were laid out already.
Procedure `pairup(Node n)`

Begin

Let \( V \leftarrow \{ v \mid n \prec_\prec v \lor n \prec_\succ v \} \)

Let \( V^+ \leftarrow \{ v \in V \mid \chi(v) = \text{positive} \} \)

Let \( V^- \leftarrow \{ v \in V \mid \chi(v) = \text{negative} \} \)

Let \( V^0 \leftarrow \{ v \in V \mid \chi(v) = \text{none} \} \)

Let \( V^* \leftarrow \{ v \in V \mid \chi(v) = \text{mixed} \} \)

Marry pairs of opposite direction bases that are not yet married.

While \( V^+ \neq \emptyset \land V^- \neq \emptyset \) do

Select \( v_1 \in V^+ \), \( v_2 \in V^- \)

Marry \( v_1 \) and \( v_2 \)

\( V^+ \leftarrow V^+ - v_1 \)

\( V^- \leftarrow V^- - v_2 \)

od

Assign directionality to \( n \) and determine sharing

If \( V^+ \neq \emptyset \) then

\( \chi(n) \leftarrow \text{positive} \)

Share a VPTR with \( v \in V^+ \)

else if \( V^- \neq \emptyset \) then

\( \chi(n) \leftarrow \text{negative} \)

Share a VPTR with \( v \in V^- \)

else if \( V^* \neq \emptyset \) then

\( \chi(n) \leftarrow \text{mixed} \)

Share a VPTR with \( v \in V^* \)

else – only \( V^0 \neq \emptyset \)

assign-initial-directionality(n)

fi

end

Figure 34: A procedure for a bidirectional layout of a class with more than 2 parents.

In the above example, \( n_\ell = 2 \), and the expected saving was \( \lfloor n_\ell/2 \rfloor / 2 = 1/2 = 0.5 \). Note that this expectation is, as with other truly randomized algorithms, with respect to coin-tosses, rather than with respect to some a priori hypothesis on the input distribution. To our knowledge this is the first randomized algorithm in domain of compiler optimization.

The expectation depends on \( n_\ell \) rather than on \( n_\ell \). This is because a class which is repeatedly inherited several times will be assigned only one directionality. In the hierarchy of Figure 2 bidirectional layout can never lead to any saving in the number of VPTRs, since the directionality of classes \( c \) and \( d \) is the same of that of \( a \).

We should also note that the theorem gives only a lower bound on the expected saving. In the next section we will see an example showing why it might be different, and how the exact saving can be computed. In certain topologies, the expected saving can be as much as \( \lfloor n_\ell/2 \rfloor + O(\sqrt{n_\ell}) \).

5 Memory Optimization Examples

To illustrate the potential reduction in space overhead that our techniques can achieve, we introduce, in Figures 35, 36 and 38, three canonical ways that multiple inheritance may be used.

The canonical examples, presented in this section, and their variants are typical of the way that applications use inheritance. Therefore, savings, similar to what we have found in our examples, will also be found in real applications. Nevertheless, an empirical study which is outside the scope of this paper is required to validate this assumption.

Figure 35 presents multiple inheritance of distinct classes in a binary tree. In the traditional object layout scheme, an object of class \( c_{15} \) requires a total of 8 VPTRs. Each node can share its VPTRs with at most one base class. A lucky assignment of directionality to classes \( c_1 \), \( \ldots, c_8 \) would reduce that number to as little as 4, which represents a 50% reduction in the compiler generated fields. By Theorem 1, the expected number of VPTRs in case directionality is assigned at random is at most 6 (a 25% reduction).

In fact, a more careful counting reveals that the expected saving is slightly greater: in nodes \( c_9 \), \( \ldots, c_{12} \) one VPTR is saved.
with probability 0.5, i.e., an expected saving of 0.5 VPTR in each. In nodes $c_{13}$ another VPTR is saved, if

$$\chi(c_1) = \chi(c_2) = -\chi(c_3) = -\chi(c_4).$$

The probability of this happening is $2 \cdot 0.25 \cdot 0.25 = 0.125$. The same expected saving occurs at node $c_{14}$. Finally, if

$$\chi(c_1) = \chi(c_2) = \chi(c_3) = \chi(c_4)$$

then a VPTR is saved at $c_{15}$. The probability of this happening is

$$2 \cdot \frac{1}{16} \cdot \frac{1}{16} = \frac{1}{128}.$$

By the linearity of expectation we have that the total expected saving in the number of VPTRs is

$$4 \cdot \frac{1}{2} + 2 \cdot \frac{1}{8} + \frac{1}{128} = 2.2578125,$$

i.e., an expected saving of 28%.

If all inheritance links in Figure 35 where virtual then the traditional model requires 49 compiler-generated fields: 15 VPTRs (one for each class as no sharing is allowed), and 34 VBPTRS. Applying eliminate-single-VI, and then bidirectional layout, may bring this number down to four.

Figure 36 presents a typical use of shared inheritance to model programming with interfaces. The inheritance hierarchy forms a ladder (in this instance with three steps) where there is an implementation inheritance of $c_1$, $c_2$ and $c_3$ and an interface inheritance of $i_1$, $i_2$ and $i_3$ such that the inheritance between implementations and interfaces, and interfaces and interfaces is virtual. The shared inheritance prevents an interface from being represented multiple times in an object of any derived class. In the traditional object layout scheme the overhead of multiple inheritance an object of class $c_9$ requires 10 compiler generated fields: 4 VPTRs and 6 VBPTRS (one of which is inessential).

![Figure 36: An interface-implementation class hierarchy.](image)

Our new techniques reduce this overhead by 80% to 2 compiler-generated fields: one VPTR and VBPTRS. The layout which achieves this is depicted in Figure 37. This was obtained by inlining $i_1$ into $i_2$, inlining $i_2$ into $i_3$, and assigning $\chi(i_1)$ ← negative, $\chi(c_1)$ ← positive. Incidentally, this layout is very similar to the bidirectional optimized layout proposed in [18] for the Theta programming language. The differences are that our layout optimization techniques are general purpose, whereas the semantics of multiple inheritance in Theta is that only single inheritance is allowed for implementation inheritance.

Finally, Figure 38 presents a portion of the class hierarchy of the C++ standard I/O library.

In the traditional object layout scheme, an object of class $c_7$ has 11 compiler-generated fields: 5 VPTRs and 6 VBPTRS (two of which are inessential). The two inessential VBPTRS point from $c_6$ and $c_5$ to $c_1$.

Applying our techniques we see that $c_7$ can be laid out using only 4 compiler-generated fields: 2 class table pointers and 2 virtual base pointers as illustrated in Figure 39.

![Figure 37: An optimized layout of class $c_9$ of Figure 36.](image)

![Figure 38: A double diamond class hierarchy.](image)

![Figure 39: An optimized layout of class $c_7$ of Figure 38.](image)
Table 2: Comparison, by example, of the number of compiler-generated fields required for the tradition and the optimized object layout schemes.

<table>
<thead>
<tr>
<th>Example</th>
<th>Fig.</th>
<th># compiler-generated fields</th>
<th>traditional</th>
<th>optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diamond</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Binary Tree</td>
<td>35</td>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Virtual Binary Tree</td>
<td>35</td>
<td>49</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Interface Implementation</td>
<td>36</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Double Diamond</td>
<td>38</td>
<td>11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Virtual Double Diamond</td>
<td>38</td>
<td>24</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Virtual η-Chain</td>
<td>10</td>
<td>2n</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Virtual η-Double Chain</td>
<td>21</td>
<td>(n^2 - n + 2)</td>
<td>2n - 1</td>
<td></td>
</tr>
</tbody>
</table>

This 63% reduction is made possible by the following optimization steps:

1. **Inlining \(c_1\) into \(c_3\).** This step represents a saving of four compiler generated fields: one V PTR (due to the sharing of a V PTR between \(c_3\) and \(c_1\)); one essential V PTR, pointing from \(c_3\) to \(c_1\); and, two inessential V PTRs, pointing from \(c_5\) and \(c_6\) to \(c_1\).

2. **Inlining \(c_4\) into \(c_6\).** This step makes it possible to share a V PTR between \(c_4\) and \(c_6\), and to eliminate the essential VB PTR from \(c_6\) to \(c_4\) for a total saving of two compiler generated fields. Note that we could also have attributed to this step the saving of the two inessential VB PTRs, which were accounted for in the previous step.

3. **Assigning directionalities to classes.** In particular, we have assigned \(\chi(c_1)\) \(\leftarrow\) positive, which imposed the same directionality on classes \(c_3, c_4, c_5\) and \(c_7\). We also assigned \(\chi(c_2)\) \(\leftarrow\) negative and \(\chi(c_6)\) \(\leftarrow\) negative. This made it possible to bidirectionally layout \(c_2\) and \(c_3\) in \(c_4\) whereby saving one more V PTR.

The frugal object layout of Figure 39 looks even more impressive when considering that it also implements the hierarchy in which all inheritance links in Figure 38 are made virtual. In this hierarchy, which represents a typical use of shared inheritance for extensible frameworks, the tradition model requires 26 compiler-generated fields for an object of class \(c_7\); 7 V PTRs, one for each class, and 19 VB PTRs, out of which 8 are essential.

We end this section with Table 2 which summarizes the savings that our optimization techniques in each of the major hierarchy examples used so far in this paper.

6 Related Work

There are not too many previous attempts to optimize object-space in C++ or in any other language which uses multiple inheritance. Burke et al. [3] present a technique to unidirectionally share a V PTR with a virtual base. A necessary condition for the applicability of their technique is that either the virtual base or the derived class have no data members. Sharing V PTR with a virtual base is another means for augmenting the traditional object layout scheme. It is orthogonal to the techniques presented in this paper, and would be easily incorporated into the set of techniques presented here.

Bacon [2] presented an algorithm for devirtualization. As explained above, that algorithm produced erroneous result when presented with duplicated classes.

A variant to the traditional scheme specifies that the V PTR is placed after the data members of the first class with virtual functions. This variant allows unidirectional sharing, but does not admit bidirectional layout, which is one of our two main optimization techniques.

Pugh and Weddell [20] present a bidirectional record layout algorithm that allows a fixed offset to be assigned to each field of a record in higher-order polymorphic (i.e., classless) programming languages with multiple inheritance. Their methods do not directly apply to our work (or the traditional way that multiple inheritance is implemented in C++ [23]) since they do not allow a subobject to start at a different offset than its containing object. In addition, in their approach fields, not objects, are assigned direction.

A more related work is that of Myers [18] who used bidirectional in the context of the Theta programming language, whose multiple inheritance semantics is similar to that of Java. Therefore, this work is restricted in scope compared to ours. In particular, only class tables are laid out bidirectionally, while unidirectional layout is still used for objects. Furthermore, all multiple inheritance is assumed to be nonvirtual. Myers proposes several strategies of dealing with the problem of direct access to a subobject: (i) restricting the language semantics so that in effect all data members are private in the C++ jargon. This is tantamount to the demand that a class designer provides special methods to access data members, if these data members might be used by inherited classes. (ii) access to all data members is always done by variables is by indirectness with offsets stored in the class table. (iii) a hybrid approach using both (i) and (ii). A time penalty in accessing data members is incurred in all of these.

7 Further Research

The two main optimization methods and we have seen that significant reductions in the number of compiler generated fields can be achieved by thoughtfully applying:

It is a matter of further research to develop algorithms for algorithms for

It is also interesting to explore other directions for space optimization. For example, it is possible to optimize further the classical diamond example, beyond what was presented here by using a technique called embedding of virtual bases.

For example, a library provides a set of classes where each class supports a service. The library makes no assumption of how the services will be used by any application. In particular, an application may decide that an application class provides services from multiple library classes.

---

4The GNU CC compiler version 2.8.1 [22] lays objects out in this way.
We hope that this work will help effect a bootstrapping process that will deepen the usage of multiple inheritance. With the reduced overhead software designers will extend their use of this feature to better support the modeling of systems forever increasing in their complexity. At the same time algorithm designers and the compilers construction community will use the patterns of increased usage in their strive to find more and more efficient implementations, which in turn will further reduce the reluctance of using this linguistic feature. We present a toolbox of techniques which In order to understand the use of

This belief was confirmed by a recent work [26] investigating the space overhead of several alternative C++ implementations of multiple inheritance. Curiously, when the authors of that work (which included one of us) went in their search for benchmarks, they discovered that there were not very many programs with abundant use of multiple inheritance. At another study of multiple inheritance [13], only one C++ application which uses multiple inheritance was found. In this application, only 1% of the classes used multiple inheritance, and none of these inherited from more than two parents.

These findings are perhaps explained by advice offered by industry leaders (e.g., [14]) against sweeping use of multiple inheritance due to the overhead it incurs.

Another evidence for this is given by the some of the examples presented in this paper which show a surprisingly large overhead for relatively simple hierarchies. For example, in a hierarchy of \( \eta \) classes, the number of compiler generated fields in certain objects can

- Who cares about multiple inheritance.
- No Experimental Evaluation

Theory has to come before experimental research. Enough work dealing with theory, believe don’t have to have evaluation. Point to appendices.

We introduce a set of canonical examples. Any inheritance hierarchy that contains multiple inheritance will be some composition of the canonical examples; that is, an inheritance hierarchy can be decomposed into components where each component is some variant of one of the canonical examples. Section 5 shows the savings for each canonical example. The savings for the entire inheritance hierarchy will be at least as much as the sum of the savings for the individual components. What are the set of components that any inheritance hierarchy can be decomposed into? State example of savings for the canonical examples. One benchmark, IDL, indicates that the amount overhead to implement object oriented features can be significant, 100increase in size of objects.

Lack of benchmarks. Chicken and egg problem.

- Has been done in the past: One of our techniques has been claimed in the past; however, that claim was erroneous. In Section 3, we provide the correct algorithm.

- (Who cares, it is not going to affect my program) small improvements in applications

- Many of the techniques require global analysis. be as large as quadratic in \( \eta \).

8 Discussion

Our optimization techniques were based on two principal ideas:

- the inlining of virtual bases which enabled a saving in the number of VBPTRs, and bidirectional object layout which gives rise to saving in the number of VPTRs. Inlining of virtual bases, a whole program analysis process, required preprocessing by elimination of transitive edges. We presented two versions of an algorithm for inlining virtual bases. The simple algorithm is guaranteed to run in polynomial time and is practice expected to run in \( O(n^2) \). This algorithm should be run in tandem with devirtualization. The more sophisticated algorithm has an exponential running time, although in practice, the fact that inheritance hierarchies tend to be sparse graphs, gives reasons to believe that in practice, this time reduces to polynomial. The sophisticated algorithm subsumes devirtualization.

Bidirectional layout which can be run in a separate compilation environment, comes in two varieties. The first variety, which we called ephemeral marriage of virtual bases is targeted at the shared variant of multiple inheritance. The expected saving of this Persistent marriage of virtual bases, the second variety is more suitable for repeated multiple inheritance.

We have presented the basic techniques, demonstrate their potential impact, and as usual in the study of algorithms, give when possible theorems to bound their worst case performance. We have seen that significant reductions in the number of compiler generated fields can be achieved by thoughtfully applying our two main optimization techniques: inlining of virtual bases and bidirectional object layout.

To take a broader perspective on this work, we note that two of the most important features provided by object-oriented languages language features require run-time support: Dynamic binding means that the method invoked in response to a message send is determined at run-time. As shown in it is next to impossible to eliminate run-time support for this language feature. Similarly, inclusion polymorphism [4] means that an object of a certain type can be addressed at run time as being of its super-type. However, it must be necessary to be able to determine the object original type.

The language feature information that is used by the run-time system may be stored in objects, as compiler-generated fields, or in tables associated with classes. However, even when information is stored in class tables, any object that needs that information must have a compiler-generated field to access the information from the class table. In C++, this pointer is called a VPT.

The overheads due to dynamic binding and inclusion polymorphism are minimal when there is only single inheritance: Each object contains exactly one compiler generated field, method addresses are stored exactly once in each class table, no indirection is involved in accessing data members, up-and down-casting involve no operation, and method dispatch involve only a simple table lookup.

The combination of multiple repeated inheritance and these two features increase the incurred overheads significantly. The
number of compiler generated fields per object becomes non-
constant, a class may store multiple copies of a method address,
casting may require a “this adjustment” and and dispatch be-
comes more complicated. Overheads increase even more with
shared multiple inheritance, since even data member access and
casting may require redirection.

The challenge of implementation object oriented language is
to minimize overheads of multiple inheritance of the shared and
the repeated kind, bringing them as close as possible to those
of single inheritance. The techniques presented in this paper
are concrete means to be used in meeting this challenge. For
the techniques presented in Section 3 the cost is in increased
compiler run time, and in the need for global program informa-
tion. These tradeoffs are not incurred with the techniques of
Section 4.

Our optimization techniques were based on two principal
ideas: the inlining of virtual bases which enabled a saving in
the number of VPTRs, and bidirectional object layout which
gives rise to saving in the number of VpTRs. Inlining of vir-
tual bases, a whole program analysis process, required prepro-
cessing by elimination of transitive edges. We presented two
versions of an algorithm for inlining virtual bases. The simple
algorithm is guaranteed to run in polynomial time and is prac-
tice expected to run in $O(n^2)$. This algorithm should be run
in tandem with devirtualization. The more sophisticated algo-
rithm has an exponential running time, although in practice, the
fact that inheritance hierarchies tend to be sparse graphs, gives
reasons to believe that in practice, this time reduces to poly-
nomial. The sophisticated algorithm subsumes devirtualization.

Bidirectional layout which can be run in a separate compi-
lation environment, comes in two varieties. The first variety,
which we called ephemeral marriage of virtual bases is targeted
at the shared variant of multiple inheritance. The expected sav-
ing of this Persistent marriage of virtual bases, the second vari-
ety is more suitable for repeated multiple inheritance.

We have presented the basic techniques, demonstrate their
potential impact, and as usual in the study of algorithms, give
when possible theorems to bound their worst case performance.
We have seen that significant reductions in the number of com-
piler generated fields can be achieved by thoughtfully applying
our two main optimization techniques: inlining of virtual bases
and bidirectional object layout.

To take a broader perspective on this work, we note that two
of the most important features provided by object-oriented lan-
guages language features require run-time support: Dynamic
binding means that the method invoked in response to a mes-
sage send is determined at run-time. As shown in it is next
to impossible to eliminate run-time support for this language
feature. Similarly, inclusion polymorphism [4] means that an
object of a certain type can be addressed at run time as being
of its super-type. However, it must be necessary to be able to
determine the object original type.

The language feature information that is used by the run-
time system may be stored in objects, as compiler-generated
fields, or in tables associated with classes. However, even when
information is stored in class tables, any object that needs that
information must have a compiler-generated field to access the
information from the class table. In C++, this pointer is called
a VPTR.

The overheads due to dynamic binding and inclusion poly-
morphism are minimal when there is only single inheritance:
Each object contains exactly one compiler generated field,
method addresses are stored exactly once in each class table,
no indirection is involved in accessing data members, up-and
down-casting involve no operation, and method dispatch in-
volve only a simple table lookup.

The combination of multiple repeated inheritance and these
two features increase the incurred overheads significantly. The
number of compiler generated fields per object becomes non-
constant, a class may store multiple copies of a method address,
casting may require a “this adjustment” and and dispatch be-
comes more complicated. Overheads increase even more with
shared multiple inheritance, since even data member access and
casting may require redirection.

The challenge of implementation object oriented language is
to minimize overheads of multiple inheritance of the shared and
the repeated kind, bringing them as close as possible to those
of single inheritance. The techniques presented in this paper
are concrete means to be used in meeting this challenge. For
the techniques presented in Section 3 the cost is in increased
compiler run time, and in the need for global program informa-
tion. These tradeoffs are not incurred with the techniques of
Section 4.

Acknowledgments The first author pays tribute to Virginia
Plowman for insightful discussions and her valuable comments
on the manuscript. Section Further Research conclusions

References

The Java Series. Addison-Wesley, 1996.

Object-Oriented Languages. PhD thesis, University of Califor-

evaluating space and time overhead for c++ object models.
Research Report RC 20421, IBM, TJ Watson Research Center,
March 1996.

tions, and polymorphism. ACM Comput. Surv., 17(4):471–522,
1985.

ple inheritance essential to OOP? Panel discussion at the Eighth
Annual Conference on Object-Oriented Programming Systems,
Languages, and Applications (OOPSLA’95) (Washington, DC),


A Counting Theorems

A.1 Computing the Number of vptrs

Let \( x \) be an arbitrary class. One way of counting the number of vptrs that \( x \) uses is to lay out an \( x \) object, as we have done above for our simple examples. However, for larger examples, this task tends to be error prone and laborious. Instead, we develop a mechanism to count the number of vptrs (and later on other kinds of memory overhead) directly from graph theoretical properties of the subobject graph \( G = G(x) \). Recall the definition of forest presented in Section 4.

Clearly, \( \mathcal{T} \) is a spanning forest since the only nodes in \( G \) with indegree greater than one are the virtual ones. Let \( n_v \) be the number of virtual nodes in \( G \). Then, \( |\mathcal{T}| = |\mathcal{E}| + \infty \), since \( x \) and each of the virtual nodes is a root of a tree in \( \mathcal{T} \).

Theorem 2 Objects of class \( x \) will be laid out by the traditional scheme using \( n_v \) vptrs.

Proof: The main idea of the proof is that the node \( x \), and any of the \( n_v \) virtual base nodes of it cannot share a vptr with any of its descendants, and therefore, it contributes at least one vptr to the total. More precisely, in the trees of \( \mathcal{T} \), which are all rooted at these nodes, vptrs sharing cannot occur between distinct paths leading from a root to a leaf. The details of the proof are omitted.

We demonstrate the theorem first by three quite simple examples. Much more elaborate cases will be presented below in Section 5. In the subobject graph Figure 4, we can infer that the number of vptrs in class \( e \) is 2 since there is only one tree in \( \mathcal{T} \) which has two leaves. Also, in our virtual inheritance example, Figure 6 is broken into two trees by removing the edges incoming to \( \alpha \); one tree is the node \( \alpha \) itself, while the other consists of all the rest. The number of vptrs is therefore \( 2 + 1 = 3 \). As a third example, consider the hierarchy of Figure 8, which gives a case of single, virtual inheritance (which therefore cannot be shared.) From its subobject graph (which can be inferred with a minimal stretch of imagination) we have that the number of vptrs in the layout of class \( x \) is \( 1 + 1 + 1 = 3 \).

Theorem 2 shows that sharing cannot occur between trees in \( \mathcal{T} \) and can not occur within a tree between distinct paths. If there is no virtual inheritance, then \( \mathcal{T} \) has only one tree, and the value of \( n_v \) is the number of leaves in this tree. Note that whenever a tree is broken into two the total number of leaves will either be the same or increase but never decreases; therefore, the number of vptrs tends to grow with the number of virtual nodes. It is not difficult to see that the number of vptrs is \( \Theta(n) \) where \( n \) is the number of nodes in the subobject graph. There are no more than \( n \) vptrs in total since each subobject has at most one vptr. There could be as many as \( n \) vptrs, in the case of a virtual inheritance chain of \( n \) nodes (depicted in Figure 10).

A.2 Counting vBPTRs

A virtual edge in a subobject graph is an incoming edge to a virtual base (those nodes whose label is encircled). Although every essential vBPTR corresponds to a virtual edge, there are virtual transitive edges which do not have a corresponding essential vBPTR. A transitive edge is a virtual edge that leads from a descendant to a virtual base which is reachable from that descendant via an alternative path. More precisely, let \( \langle \alpha, \gamma \rangle \) and \( \langle \beta, \gamma \rangle \) be edges in the subobject graph such that \( \alpha \), the class of \( \alpha \), inherits from \( \gamma \), the class of \( \beta \). Then, there must be a path from \( \alpha \) to \( \beta \), and therefore the edge \( \langle \beta, \gamma \rangle \) is transitive; the containment relationship between the \( \alpha \) subobject and the \( \gamma \) (virtual base) subobject is maintained via a path through \( \beta \). A vBPTR corresponding to the transitive edge \( \langle \alpha, \gamma \rangle \) cannot be essential, since access from the \( \alpha \) subobject to its \( \gamma \) virtual sub-object is possible without it, e.g., through a vBPTR corresponding to \( \langle \beta, \gamma \rangle \).

Let the essential subobject graph be the graph obtained from the subobject graph by removing from it all transitive edges. For an example see Figure 40.

Almost by definition we have

Theorem 3 The number of essential vBPTRs in the layout of class \( x \) is the number of virtual edges in its essential subobject graph.

Let \( m \) denote the number of edges in the essential subobject graph. Then, we can also assert:

Theorem 4 Objects of class \( x \) are laid out by the traditional scheme using \( n_v + m - n_v \) essential vBPTRs.

Proof: Let \( \mathcal{T} \) be the same as in Theorem 2. For \( T \in \mathcal{T} \), let \( T_n \) denote the number of nodes of \( T \) and \( T_m \) the number of its edges. Since \( \mathcal{T} \) is a tree we have

\[
T_n = 1 + T_m.
\]

Summing the above over all \( T \in \mathcal{T} \), and noting that \( |\mathcal{T}| = |\mathcal{E}| + \infty \), we obtain

\[
\sum_{T \in \mathcal{T}} T_m = n - n_v - 1
\]

which can be written as

\[
\sum_{T \in \mathcal{T}} T_m = n - n_v - 1
\]

The essential vBPTRs are exactly the edges of \( G - \mathcal{T} \). Therefore, their number is \( m - \sum_{T \in \mathcal{T}} T_m \). The theorem now follows by substituting the above equation into this expression.

Let us break the expression \( n_v + m - n_v + 1 \) in the above theorem into two terms: \( n_v \), the number of virtual bases, and \( m - n_v + 1 \) which is nothing but the number of edges that have to be removed from the essential subobject graph in order to make it a tree. Curiously, these two terms correspond to two kinds of essential vBPTRs. The primary
essential VBPTRs are those which are required to reach from \( x \) any of its virtual bases. Clearly, there must be exactly \( n_0 \) primary essential VBPTRs. In contrast, the secondary essential VBPTR are those which are needed by some of the proper subobjects of \( x \), and as the theorem indicates, there are exactly \( m - n + 1 \) of those.

The chain example above (Figure 10) showed that the number of VBPTR is \( \Omega(n^2) \) in a class whose subobject graph has \( n \) nodes. In fact, even the number of essential VBPTRs is \( \Theta(n^2) \). To see this observe that there could never be more than \( n^2 \) VBPTRs, since each subobject can have at most one VBPTR to any other subobject. For the lower bound side, consider the subobject graph, depicted in Figure 41 of a class \( c \) which has also nodes \( a_1, \ldots, a_m, b_1, \ldots, b_n \), and edges between each \( b_i, i = 1, \ldots, n \), and each of the \( a_j, j = 1, \ldots, n \), as well as edges between \( c \) and each \( b_i, i = 1, \ldots, n \).

The number of essential VBPTR is never greater than the number of edges in \( G \). In order to compute the total number of VBPTR, which could be much greater than that, we need to compute the transitive closure of this graph. This operation is, in a sense, the opposite of the construction of the essential subobject graph.

Give a subobject graph \( G \), we construct from it a multi-graph \( G^* \) as follows: We first compute the essential subobject graph \( G_x \), from which we create the multi-graph \( G_T \), in which the nodes are the trees of \( T \), and there is an edge between trees \( T_1 \) and \( T_2 \) for every edge that connects in \( G_x \) between a node of \( T_1 \) and a node of \( T_2 \). Then, \( G^* \) is computed as the transitive closure of \( G_T \). Let \( m^*_0 \) be the number of edges in \( G^* \).

**Theorem 5** Objects of class \( x \) will be laid out by the traditional scheme using a total of \( m^*_0 \) VBPTRs.

**Proof:** Omitted.

As a corollary we obtain that the number of inessential VBPTR is

\[
m^*_0 = m + n - n_0 - 1.
\]

This is exactly the space overhead of the C++ semantical requirement that an upcast to a virtual base must be performed in constant time.

**B Families of Optimization Algorithms**

After having presented our basic optimization techniques in Sections 3 and 4, and having demonstrated the significant reductions in the number of compiler generated fields that these may bring about Table 1, it is time to ask how does one determine in which locations in an inheritance hierarchy these basic techniques are to be applied in order to maximize their effectiveness. Exploring the answers to this question is the subject of this appendix.

The basic techniques of eliminating transitive virtual inheritance edges and devirtualizing inheritance edges is completely closed. There is no room for variations in their implementation. Unlike these, the procedure for inlining virtual bases presented in Section 3.3 left open the selection of an immediate virtual descendant into which inlining was to be done. Similarly, the marriage of virtual base classes could be done in many different ways. In the same fashion, since the initial assignment of directionality to roots of the inheritance hierarchies is arbitrary, it is possible in principle to carefully choose these assignments in attempt to achieve better performance of the bidirectional layout algorithm. Moreover, since our techniques modify the structure of the inheritance hierarchy, they are interdependent. Decisions made in one algorithm could affect the performance of others.

This section explores the topic of algorithm families. Section B.1 presents the notion of class weights—which serve as the most important means for evaluating the performance of layout algorithms. In Section B.2 we go into into length to list the members of a family of algorithms for the problem of inlining virtual bases. Section B.3 gives an alternative strategy for choosing the initial assignments of directionality to classes. Finally, in Section B.4 we propose a master-algorithm which choose particular members of this families in attempt to balance between level of optimization and consumption of computational resources. This master algorithm is the one used in making the optimizations in the examples of the previous section.

**B.1 Class Weights**

We have seen how the the selection of a descendant for inlining is made.

The result of inlining virtual bases is at least as good as that of the traditional object layout scheme. This is because inlining never increases the size of any class. It always saves one essential VBPTR; it always makes the inlined virtual base a candidate to share its VPTR, with the descendent it is inlined into, or with a sibling via bidirectional layout, which may lead to a reduction in the number of class table pointers that are required. In addition to all these, inlining may also save one or more inessential VBPTRs.

If information on the relative frequencies of class instantiations is available, we may be able to use it to tune the algorithm to favor inlining into classes which are instantiated more than others. When no such run-time information is available, it seems plausible to apply the heuristic of inlining in such a way that would maximize the number of classes that benefit from it.

To make these ideas more rigorous, we introduce the function \( p(x) \) which serves as a metric of the number of direct instantiations of \( x \). Counting instantiations is a very illusive prospect: It could be done on a specific run of the program on one specific input, an average (arithmetical, geometrical, or of other sorts), weighted or unweighted over a sample of runs, or over the whole ensemble of all runs, or perhaps the maximum over such sample or the entire ensemble. Another way of defining \( p(x) \) is as a high-water mark, i.e., the maximal number of co-existing instances of \( x \). Again, this reading could be taken on one run, or as a combination of some sort of more than one run.

With the absence of run-time information, a definition of \( p(x) \)
basis on static information may be

\[ \rho(x) = \begin{cases} 0 & \text{if } x \text{ is an abstract class} \\ 1 & \text{otherwise} \end{cases} \]

Such definition represents what we may call the zero hypothesis in which each class is equally likely to be instantiated. One can imagine more elaborate static-based definitions of \( \rho(x) \), based for example on a heuristic that the the greater the number of descendants a class has, the less likely it is to be instantiated.

More fundamentally, regarding \( \rho(x) \) as a weight function is only a heuristic, because there are dependencies between the instantiations of different classes. Although a certain class \( z_1 \) may have the same number of instantiations as another class \( z_2 \), it could make more sense to invest our saving efforts in \( z_1 \) if it tends to be instantiated in circumstances where memory is scarcer than when \( z_2 \) is instantiated.

Although it is possible in principle to tune our algorithms to, say, minimize the maximal space overhead of a program averaged over all runs, but the resources for running such a massive optimization effort would be enormous. We are inclined to follow the standard practice of research on performance optimization and computer engineering, and optimize against some kind of a metric function, as inaccurate and as incomplete it may be. The exact definition of the function \( \rho \) will be left open. In other words, our algorithms and results are parameterized by a selection of a definition of \( \rho \) as an appropriate heuristic for the actual needs and the availability of run-time performance information. Nevertheless, remembering the point made here that \( \rho \) is basically a heuristic, we will allow ourselves, with reluctance, to fall short of the ultimate optimization with respect to \( \rho \). The same latitude could be obviously taken when optimizing with respect to any other metric function derived from \( \rho \).

Based on any definition of \( \rho \) we can define a metric of the number of direct and indirect instantiations of a class. For a class \( x \), which might even be abstract, let \( \omega(y) \) be the number of times that \( y \) is instantiated, either as an object or as a subobject,

\[ \omega(x) = \sum_{y \in L^e} \rho(y) \mu(y)(x), \]

where \( \mu(y)(x) \) is the number of times that a subobject of type \( y \) occurs in an object of type \( x \) (subscript \( y \) is not used when it is clear from context).

The weight function \( \omega \) is more meaningful than \( \rho \). For example, while debating on whether a particular virtual base should be inlined into \( u_1 \) or into \( u_2 \), it should taken into account that this decision has also an impact on the layout of classes derived from \( u_1 \) or \( u_2 \). This impact is concisely captured by the values \( \omega(u_1) \) and \( \omega(u_2) \). Any savings in the size of \( u_i, i = 1, 2 \) are multiplied by the factor \( \omega(u_i) \). Consequently, \( \omega \) will be our main optimization function.

**B.2 A Family of Algorithms for Inlining Virtual Bases**

The primitive technique of inlining virtual bases engenders not one but a family of algorithms which use weights to determine how this technique might be applied. These algorithms represent a battery of optimization heuristics. Together, they span a full spectrum of memory savings prospects by offering a tradeoff between the demand they place on compiler resources and how close they come to optimal layout with respect to the weight function. This subsection enumerates the members of this family in an approximate order ranging from the most modest to most massive optimization effort, while leaving it to further research, of the experimental engineering kind, to explore the suitability of each family member for any given real life situation.

1. The simplest algorithm for inlining virtual bases is that presented in Figure 22 of Section 3.3. The selection of an immediate descendant is done arbitrarily, and can be implemented with no run-time information. As mentioned above, such a selection can even be implemented in separate compilation contexts.

2. At the next level of our optimization effort, we can use the function \( \omega \) in the selection of the descendant into which inlining is made, favoring the one which maximizes this function. Computing the value of \( \omega \), given those of \( \rho \) can be done in time linear in the size of the inheritance hierarchy. Such linear time is also what is required in total for descendant selection for all virtual bases in the hierarchy.

   In assuming that \( \rho(x) = 1 \) for all \( x \), we have that in the interface-implementation hierarchy (Figure 36) of Section 5, \( \omega(z_1) = 4 \) whereas \( \omega(z_2) = 3 \). This explains our decision of inlining \( z_1 \) into \( z_2 \) rather than into \( z_2 \).

3. Beyond the guaranteed saving of one essential \( \text{VBPTR} \), inlining has also the potential of multiple savings of inessential \( \text{VBPTRs} \). Specifically, if \( u_1 \prec u_2 \), then by inlining \( u_2 \) into \( u_1 \), two kinds of savings of inessential \( \text{VBPTRs} \) are made possible.

   1. An outgoing inessential \( \text{VBPTR} \) from \( u_1 \) to a node \( v \), \( v \prec_\omega u \) is saved if the path of inheritance leading from \( u_1 \) to \( u_2 \) ends with a virtual inheritance edge which was not inlined, and all other edges in this path are either nonvirtual or already inlined. The benefits of savings of outgoing inessential \( \text{VBPTRs} \) are modeled by the function \( \omega(u_1, v) \) which is defined as \( \omega(u_1) \times \omega(v) \) times the number of such \( \text{VBPTRs} \).

   2. Also, an incoming inessential \( \text{VBPTR} \) from a node \( u \) (and each of its descendants) to \( u_2 \), \( u_2 \prec u \) is saved, if the path of inheritance leading \( u_1 \) and \( u_2 \) starts with a virtual inheritance edge which was not inlined, and all other edges in it are either nonvirtual or already inlined. Similarly, the benefits of savings of incoming inessential \( \text{VBPTRs} \) are modeled by the function \( \omega(u_1, u) \) which is defined as the sum of \( \omega(u_1) \times \omega(u) \) for all such \( v \).

   We should therefore in-line \( u_2 \) into its descendant \( u_1 \) which maximizes the objective function

   \[ \phi(u) = \omega(u) + \omega(v) + \omega(u, v) \]

   The problem in doing so is that the functions \( \omega \) and \( \omega(v) \) are strongly dependent on each other for different \( v \)’s. As we have seen in the double diamond example of the previous section, the attribution of the savings of inessential \( \text{VBPTRs} \) to specific inlines is rather ambiguous. The outgoing \( \text{VBPTRs} \) of one pair of \( u_1 \) and \( u_2 \) are always incoming \( \text{VBPTRs} \) of another such pair. Worse, one inlining decision could open, or block, opportunities for savings of inessential \( \text{VBPTRs} \) in another inlining context. Thus, inlining decisions of different virtual bases are not independent and, if absolute optimization is desired, must be done all together. Trying to do so might make the optimizer run in a time exponential in the size of the inheritance hierarchy.

   We propose a heuristic in which the inheritance hierarchy is traversed in a topological-sort order, starting at the inheritance roots and ending with classes with no descendants at all. For each virtual base encountered during the traversal, we chose a descendant to inline into
based on function $\phi$. After each such decision, the values of the functions $\overline{\omega}$ and $\omega$ and consequently $\phi$ are updated for the remaining nodes in the hierarchy. This algorithm can certainly be implemented in time polynomial in the size of the hierarchy. We suspect that it can even be done in time $O((n + m)\log n)$, where $n$ and $m$ are the number of nodes and edges in the hierarchy, although more graph algorithmic research is required to verify this.

An alternative heuristic is to apply this process in a reverse order. Our intuition is that this alternative will not lead to superior results.

The careful counting of inessential VPtrs explains the decision, in the example of Figure 36, to inline $i_2$ into $i_0$ rather than into $c_1$, even with the assumption that interface classes are abstract, i.e., $p(x) = 0$ for $x = i_1, i_2, i_3$ ($p(x)$ is still 1 for implementation classes). We see that $\omega(c_1) = 3 > \omega(i_2) = 2$, and $\overline{\omega}(c_1, i_1) = \overline{\omega}(i_0, i_2) = 0$. On the other hand, $\omega(c_1, i_1) = 0$, while

$$\overline{\omega}(c_1, i_1) = \omega(c_2) + \omega(c_3) = 2 + 1 = 3$$

Thus,

$$\phi(c_1) = 3 + 0 + 0 < \phi(i_2) = 2 + 0 + 3 = 5$$

Descending the hierarchy to class $i_2$, we have that the updated values are $\overline{\omega}(c_2, i_2) = 0$ for $x = c_2, i_2$, while the updated values of $\omega$ are $\omega(c_2, i_2) = 0$ and $\omega(i_3, i_2) = \omega(c_3) = 1$. We therefore have $\phi(c_2) = 2 + 0 + 0 = 2$, and $\phi(i_2) = 1 + 0 + 1 = 2$. What may tip the balance in this case is the fact that $i_2$ is the sole parent of $i_3$, which means that a VPtr is guaranteed to be saved due to sharing in case $i_2$ is inlined into $i_3$.

4 Computing the functions $\omega$, $\overline{\omega}$ and $\omega$ requires global program information. However, in the presence of such information, we might even do better, since it is possible to inline a virtual base $\psi$ into more than one of its descendants, as long the rule that no two inlined descendants share a common descendant $u$ of their own, is kept. The common descendant $u$ will create an incident of conflicting offsets of the subobject of $\psi$.

Again, any inlining strategy which observes this rule is going to be at least as good as the traditional object layout scheme. The algorithm for selecting such strategy is given in procedure inline-VB of Figure 23 that was presented in inlining.

As noted above, the maximal weighted independent set problem is known to be NP complete.

5 As before, the function $\omega$ used as the weight function in the maximal weighted independent set algorithm does not cater for savings of inessential VPtrs. It is merely a matter of a technical exercise to see that for independent sets, the functions $\overline{\omega}$ and $\omega$ are additive. We can therefore use $\phi(u)$ as the weight function.

If this is done, we must choose an order of traversing the virtual base nodes of $I$, and take care to update the weights $\overline{\omega}$ and $\omega$ as inlining decisions are made.

6 Since the savings in inlining of virtual bases are not independent, the only correct approach for finding the best inlining strategy with respect to a given weight function is an exhaustive one, i.e., trying out all possibilities of simultaneously inlining all virtual bases. In this exhaustive approach, weights are used to determine the minimum over all possible choices of inlining; given an inlining strategy, compute the number of compiler-generated fields per class and sum that number over all classes. The complexity of this approach is exponential in the number of virtual inheritance edges, which would probably be too much for most inheritance hierarchies.

We note that in all the other algorithms presented so far, the weights are used only as means of a heuristics to help the algorithm make better decisions.

7 The weight functions used so far did not capture the potential of saving by inlining that an inlined base $\psi$ may share a VPtr with its descendant $u$, into which it is inlined. This saving is guaranteed if $\psi$ is the only parent $u$ has. However, if $u$ has more than one parent, then no such saving is possible unless bidirectional layout is applied. A simple weight function that tries to model this is

$$\alpha(u) = \begin{cases} \omega(u) & \text{if } u \text{ has exactly one parent} \\ \beta \omega(u) & \text{if } u \text{ has more than one parent, and} \\ 0 & \text{bidirectional layout is applied} \end{cases}$$

where $\beta$ is an empirical constant, $0 < \beta < 0.5$ which models the probability of sharing a VPtr due to bidirectional layout. We also let $\psi(u) = \alpha(u) + \phi(u)$ to be our main objective function.

The function $\alpha$ does not model the intricate situation that occurs if $\psi$ has virtual parents or even ancestors other than $\psi$. As far as VPtr savings is concerned, it might be that in this case that refraining from inlining $\psi$ into $u$ would increase the chances for saving a VPtr due to added opportunities for ephemeral marriages. We are unable to find a good way of modeling this phenomena as a weight function, and are forced to resort to exhaustive algorithms in order to account for its effect.

**B.3 Global Pairing of Virtual Bases**

Consider for example the inheritance topology of Figure 42. If classes $u_1$ and $u_2$ do not override any virtual methods introduced in their virtual bases $v_1$, $v_2$, and $v_3$, then if $v_2$ is married with $v_1$ in $u_1$ and with $v_3$ in $u_2$, then we will need multiple virtual function tables for $v_2$ when only one would be required if no ephemeral marriage occurred.

What might be done in order to minimize this undesirable duplication of VTBLS of virtual bases is to try to preserve in a class as many as possible ephemeral marriage decisions that were made in its parents. In Figure 42, we would like the marriage decisions made in $u_1$ and $u_2$ to be preserved as much as possible in class $u_3$. A heuristic for doing so using a graph theoretic matching algorithm is described in Figure 43. A matching in a graph is a set of edges such that no two of these are incident on the same node. There are numerous efficient algorithms for matching in the literature, all of which run in polynomial time.

Note that the procedures Figure 31 and Figure 43 are just two members in a family of algorithms that could be used for making ephemeral marriage decisions. We could for example use a weighted matching algorithm in order to try to preserve marriages of classes with large
VTBLs. Further, since $|V^+|$ is often not equal to $|V^-|$, we can prefer to leave unmatched those classes whose VTBLs are the largest. At yet a higher level optimization effort, one could even forego the matching algorithm, consider all possible ephemeral marriages strategies between the members of $V^+$ and $V^-$. One could even consider together all classes $u \in H$ and find a global ephemeral marriage strategy which minimized the unnecessary duplications of VTBLs. Finally, one could even backtrack the directionality assignments in order to minimize the duplication of VTBLs due to ephemeral marriages.

This is only the first of algorithm families which represent a trade-off between the degree of optimization and the expense in terms of computational resources required for achieving this degree. Another such family which strides upon the inlining virtual bases technique is described in a great detail in Section B.2. This other family is richer than the one describe in the current section since it must also take into account the relative frequencies of class instantiations, a concept which was irrelevant here since only class-space optimization was an issue. Other families are mentioned as the discourse progresses.

As noted above, an ephemeral marriage of virtual bases does not lead to any saving in the number of VBPtrs. We now turn into exploring the possibility of using bidirectional layout for a permanent marriage between virtual bases. We will first examine the option of assigning directionality together virtual bases before.

Suppose that we have decided to permanently marry virtual bases $v_k$ and $v_j$, such that $\chi(v_k) = negative$ and $\chi(v_j) = positive$. Then, in every class in which $v_k$ and $v_j$ occur there is in essence one compound subobject which has the subobject $v_k$ in negative offsets, and the subobject $v_j$ in positive offsets. With what other similar decisions would this decision conflict? Trivially, such a decision will conflict with any decision to assign $\chi(v_k) = positive$ or with any decision to assign $\chi(v_j) = negative$. Beyond this, it would conflict with a decision to permanently marry $v_k$ and $v_j$ if there is a class $w$ which virtually inherits from all three classes, i.e., $w <_v v_k, w <_v v_j$ and $w <_v v_j$. Since, in objects of class $w$, $v_k$ could not be permanently married with both $v_j$ and $v_j$. The symmetrical case decision, in which $v_j$ is married with $v_i$ and where there is a class $w$ such that $w <_v w_i, w <_v v_j$ and $w <_v v_i$ will also raise a conflict.

We can therefore find a global strategy for permanently marrying virtual bases as follows. Construct a graph of the nodes of the form $\langle v_k, v_j \rangle$, representing the decision to permanently marry $v_k$ with $v_j$, while assigning $\chi(v_k) = negative, \chi(v_j) = negative$. The edges in this graph will represent conflicts among prospective decisions. Now we assign weights to each possible permanent decision

$$\omega(\langle v_k, v_j \rangle) = \sum_{w, w <_v v_k, w <_v v_j} \rho(w).$$

Finally, we run a weighted maximal independent set algorithm to find the best set of non-conflicting decisions, which will serve as our global strategy for permanently marrying virtual bases. The full algorithm is given in Figure 44.

A not too deep scrutiny of this process unfolds a family of algorithms here as well. For example, one could run this algorithm after the assignment of directionality has been carried out, or even together with it. Further, it might be possible to consider all these possibilities together with the different alternatives for ephemeral marriage, and even with the decisions for inlining virtual bases. However, since the basic algorithm is exponential in the square of the number of virtual bases, we will not bother to investigate this family any further.

---

Figure 43: A procedure for marrying virtual bases which tries to preserve marriage decisions made by parents.
Procedure persistent-marriages(Hierarchy $H$)

[2] Begin

[3] Let $U \leftarrow \{v : \exists u \leq v, v \}

[4] Let $V \leftarrow \emptyset, E \leftarrow \emptyset, G \leftarrow (V, E)

[5] For each $u, v \in U$

[6] Let $q \leftarrow \langle u, v \rangle$

[7] $V \leftarrow V \cup \{q\}$

[8] For each $v, \in U$

[9] $E \leftarrow E \cup \{\langle u, v \rangle \}$

[10] od

[11] For each $v \in U$

[12] $E \leftarrow E \cup \{\langle v, v \rangle \}$

[13] od

[14] For each $q' = \langle u, v \rangle \in V, q' \neq q$

[15] If $\exists w \in H, w < u, v, w < u, v$

[16] If $\exists w \in H, w < u, v, w < u, v$

[17] $E \leftarrow E \cup \{\langle q, q' \rangle \}$

[18] od

[19] fi

[20] od

[21] od

Let $S \leftarrow \max-wgt-indep-set(G)$

For each $\langle u, v \rangle \in S$

$\chi(u) \leftarrow$ negative, $\chi(v) \leftarrow$ positive

For each $w, d < u, v, d < u, v$

Marry $u$ and $v$ in $w$

od

od

end

Figure 44: Persistent marriage of virtual bases.

Procedure master-layout-algorithm(Hierarchy $H$)

[2] Begin

[3] eliminated-extraneous-VI($H$)

Inline virtual base classes of $H$

[4] Compute weights $\omega$ and $\overline{\omega}$

[5] For each $v \in H$ in topological order do

[6] If $\exists w \in H, u \leq u, v$ then

[7] maximum-inline-VB($v$)

[8] Update weights $\omega$ and $\overline{\omega}$

[9] od

pair up virtual bases to save VPtr and VBPtrs.


Layout a classes parents first

[12] For each $\pi \in H$ in topological order do

[13] bidirectional-layout($\pi$)


[15] ephemeral-marriage($\pi$)

[16] end Layout

Figure 45: A master algorithm for laying out an entire class hierarchy.

B.4 Putting the Algorithms Together

DO WE NEED THIS SUBSECTION?? PFS

This subsection describes a master algorithm which puts all of our techniques and algorithms together. We believe that this master algorithm gives good optimization results, without resorting into exhaustive searches.

Figure 45 presents the algorithm. Several comments apply:

1. In running procedure maximum-inline-VB, we search a maximal weighted independent set using on the weight function $\psi$, with $\beta = 1/4$.

2. In maximum-inline-VB we apply an exhaustive search to find the maximal independent set, unless the number of immediate virtual descendants is greater than $\log |H|$, in which case we apply a heuristic.

3. The procedure persistent-marriage is only called if the number of virtual bases is $< \sqrt{|H|}$.

4. A heuristic is applied for finding the maximal independent set in persistent-marriage.

5. There are straightforward changes to the algorithm bidirectional-layout to cater for the fact that persistent-marriage may have assigned directionalities to classes.

6. We do not make any attempt to optimize the run of ephemeral-marriage.

With these restrictions, it is guaranteed that master-layout-algorithm runs in polynomial time in the size of $H$. 
C Todo list

1. Weave in the running example.

2. Inlining a virtual base is a global decision, and it can be used to save vbptrs: Figure double:ladder, $C_2$ needs its own vbptr to $\tau_2$, but we may be able to get by using $C_1$’s vbase pointer to $\tau_1$, when $\tau_1$ is inlined into $\tau_2$.

3. Use local marriage instead of persistent marriage between two virtual bases when: $a$ and $b$ are virtual bases, $a \prec b$, and $a$ has other base class that $b$ is also a virtual base of. This occurs in our example between classes $d \prec a$.

4. Add to the Section 6 "An Extended Model of Shared Inheritance" the argument that C++ specification of the type of inheritance on an edge forces the decision of the type of inheritance too early.

5. Breakdown of papers

   (a) OOPSLA paper: traditional object layout, techniques, and canonical examples. One appendix family of algorithms without evaluation. Another appendix, counting theorems with out proofs.

   (b) New paper for counting theorems, lower and upper bounds, proofs, apply to running example. Include Moments (count number of construction VFTs and the amount of time for construction). Figure out “Other Overhead”. Include definitions of a subobject graph, of a class hierarchy graph, and the algorithms to go from one to the other.

   (c) Extended model (letter submission). Show deficiencies in C++ object model for specifying type of inheritance: too early and fixes implementation. Suggest new model which fixes the deficiencies. Selective sharing: class $A$ : class $B$: public virtual $A$ class $C$: public virtual $A$ class $D$: public $A$, public $B$ class $E$: public $A$, public $B$ class $F$: public $D$, public $E$ where there are two $A$ subobjects in an $F$ object.

   (d) Graduate student (Yossi finds). Implement family of algorithms and evaluate on benchmarks. Time complexity and precision.