RC 21370(96227) (Revised 4/2/99)
Computer Sciences/Mathematics

IBM Research Report

Quantifying and Evaluating the Space Overhead in C++ Memory Layouts

Peter F. Sweeney, Michael Burke

IBM T.J. Watson Research Center

P.O. Box 704, Yorktown Heights, NY 10598, USA

{pfs,burkem}@watson.ibm.com

IBM Research Division
Almaden · Austin ·Beijing · Haifa · T.J. Watson · Tokyo · Zurich

Quantifying and Evaluating the Space Overhead for Alternative C++ Memory Layouts

Peter F. Sweeney and Michael Burke
IBM Thomas J. Watson Research Center

Abstract

This paper develops a formalism that precisely characterizes when class tables are required for C++ memory layouts. A memory layout is a particular choice of data structures for implementing run-time support for object-oriented languages. We use this formalism and other techniques to quantify and evaluate, on a set of benchmarks, the space overhead for a set of C++ memory layouts. In particular, this paper studies the space overhead due to three language features: virtual dispatch, virtual inheritance, and dynamic typing. To date, there has been no scientific quantification or evaluation of C++ memory layouts. Our approach can help C++ implementors. This work has already influenced the memory layout design choices in the next version of IBM's Visual Age C++ compiler.

Applying our approach to a set of five benchmarks, we demonstrate that the impact of object-oriented space overhead can vary dramatically between applications (ranging from 0.42% to 99.79% for our benchmarks). In particular, applications whose object space is dominated by objects that heavily use object-oriented language features will be significantly impacted by the choice of a memory layout. The two end points in the spectrum of our memory layouts differ by at least 95% for each of our benchmarks.

1 Introduction

Object-oriented languages in general, and C++ in particular, provide language features that require run-time support. The usual implementation of the run-time support maintains information that is stored in objects and/or in tables associated with classes. A particular choice of data structures to represent this information is termed a *memory layout*. Each C++ compiler implements a choice of data structures. To date there has been no scientific quantification or evaluation of these different choices. This paper is concerned with measuring the size of the data structures for alternative implementations of C++ run-time support, and thereby comparing compiler implementations.

Run-time space overhead is an important consideration for C++ implementations. Virtual inheritance is needed for good object-oriented design techniques [20, 19], but conventional techniques for implementing virtual bases are inefficient in terms of space [17, 12]. Space overhead is of particular importance for space-constrained systems such as embedded and pervasive computing systems. Most previous work on efficient C++ implementation has focused either on implementing just one language feature, such as multiple inheritance [24, 22, 9, 10]; or on reducing time overhead, such as for virtual dispatch [4, 11, 3, 6, 21, 8, 2]. In this paper, we quantify and evaluate the space overhead for three language features: virtual dispatch, virtual inheritance, and dynamic typing. We also study the space overhead due to the interaction between these features.

Computing the space overhead for a memory layout requires computing the object overhead and the overhead due to class tables. For each memory layout, the information for some language features is stored as entries in class tables (class entries), and for others, the information is stored as entries in objects (object entries). Section 1.1 illustrates the placement of language features for two memory layouts. The object space overhead can be computed from the number of objects created for each class times the number of object entries in that class. The space overhead due to class tables can be computed from the number of tables required for each class times the number of class entries for that class. However, it is straightforward for each class to compute the number of objects created and, for any given memory layout, the number of object entries and of class entries. It is not straightforward to compute the set of class tables required for each class. This paper focuses on this computation. One contribution of this paper is to formalize the exact conditions under which a new class table is required.

One approach to computing the space overhead of class tables for a memory layout is to implement it. The set of class tables for each class can then simply be counted by compiling an application. This approach has two fundamental problems. An implementation does not provide a flexible framework to understand variations in memory layouts. Furthermore, the computation of space overhead is not precise unless the implementation generates a class table only when one is required. IBM's Visual Age C++ compiler [5] generates unnecessary construction class tables, because its implementation is conservative.

In contrast to the implementation approach, our approach is built upon a formalism which we call a *pattern*. A pattern is applied to class hierarchy graphs (CHGs) and specifies the exact constraints under which class tables are required. Our formalism for studying space overhead is independent of any particular memory layout in that patterns are generated with respect to a family of memory layouts. General assumptions about memory layout determine the set of relevant patterns. If a new memory layout in this family is introduced, we can use the same set of patterns to study space overhead in the new layout. Making a different set of assumptions may result in a different set of patterns, which can then be used to explore a new family of memory layouts. This paper examines a set of memory layouts that have been implemented in the industry or proposed in the literature.

Our approach can help C++ implementors to precisely identify the conditions under which class tables are required, and to understand the space overhead tradeoffs when they choose to implement a particular memory layout. This work has already influenced the memory layout design choices in the next version of IBM's Visual Age C++ compiler.

Section 1.1 illustrates contrasting choices of where to place language feature information. Section 1.2 discusses related work. Section 2 formalizes the concept of a pattern, describing how patterns are used to determine class table requirements. Section 3 presents the set of patterns for the C++ features of virtual dispatch, virtual inheritance, and dynamic type casting. Section 3.4 presents the patterns for the interaction between class entries for these language features. Section 4 presents the evaluation of the class table and object space overhead for five benchmarks. Section 5 discusses future work and Section 6 draws conclusions.

1.1 Variations on Memory Layout

The usual implementation of object-oriented language features requires information to be stored in objects and/or in tables associated with classes. Memory layouts differ in where language feature information is placed. For information stored in class tables, any object in need of that information must have a class table pointer (CT_PTR) to access it.

Consider the example class hierarchy graph in Figure 1. There are multiple ways that class $\mathcal{C}$ and its instances can be laid out in memory. For the memory layouts that we consider, the two extreme layouts are denoted `ARM` and `ALL`. In the `ARM` memory layout as much language feature information as possible is placed in objects. This approach is illustrated in Figure 2, where dynamic typing (DTE) and virtual inheritance (VIE) information are stored in objects. In the `ALL` memory layout as much language feature information as possible is placed in classes. This approach is

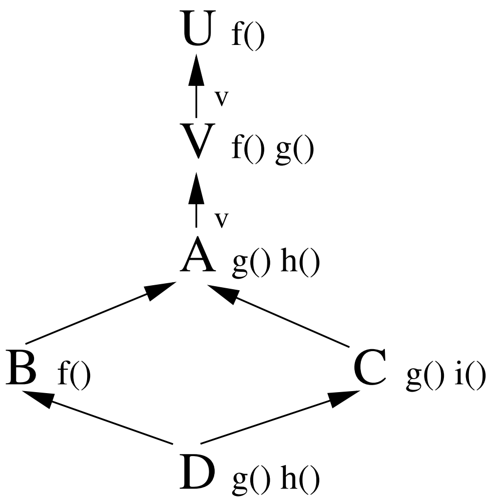

Figure 1: Example CHG

illustrated in Figure 3, where dynamic typing and virtual inheritance information are stored in class tables. For both approaches, virtual function information VFE is placed in class tables.

These figures illustrate that A and C share a class table, CT, and that the A subobject and the C object share language feature information. For ARM, the A subobject and the C object share one CT_PTR, two VIEs, and one DTE. For ALL, the A subobject and the C object share a CT_PTR. Since ALL places VIEs and DTEs in class tables and A and C share class tables, a C class table uses the one DTE and two VIEs of the A class table instead of introducing these entries separately.

For either memory layout, the same information is required: 3 CT_PTRs, 3 DTEs, 3 VIEs, and 4 VFEs for a total of 13 compiler-generated fields. Nevertheless, the ALL approach has only three compiler-generated fields in objects of class $\mathcal{C}$, while the ARM approach has nine. A class table is shared by all objects of that type. If many class $\mathcal{C}$ objects are created, the ARM approach will pay a high price in terms of space overhead; however, the cost of accessing the information is less expensive since no indirection is required. For more information on memory layouts see [18] and [12].

Time may be an important issue for some applications in a particular computing environment. This paper addresses the time tradeoff for memory layouts in Section 5.

In Section 4, three additional memory layouts will be considered and compared.

1.2 Related Work

Much of the related work has already been discussed in Section 1. Here we discuss other related work.

Past research focused on innovative and alternative memory layouts [5, 23], but did not address dynamic typing or the sharing of class information between classes, and used an approximate analytical model to compute both time and space overhead.

Recent work [12] has examined at two new memory layout techniques to dramatically reduce the object space overhead of object-oriented language features without increasing time overhead. Although promising results are reported, one technique, inlining virtual bases, requires whole program analysis, which is not always available in production compilers. The other technique, bidirectional layout, is not backward compatible with industry standards, such as Microsoft's Component Object Model [7].

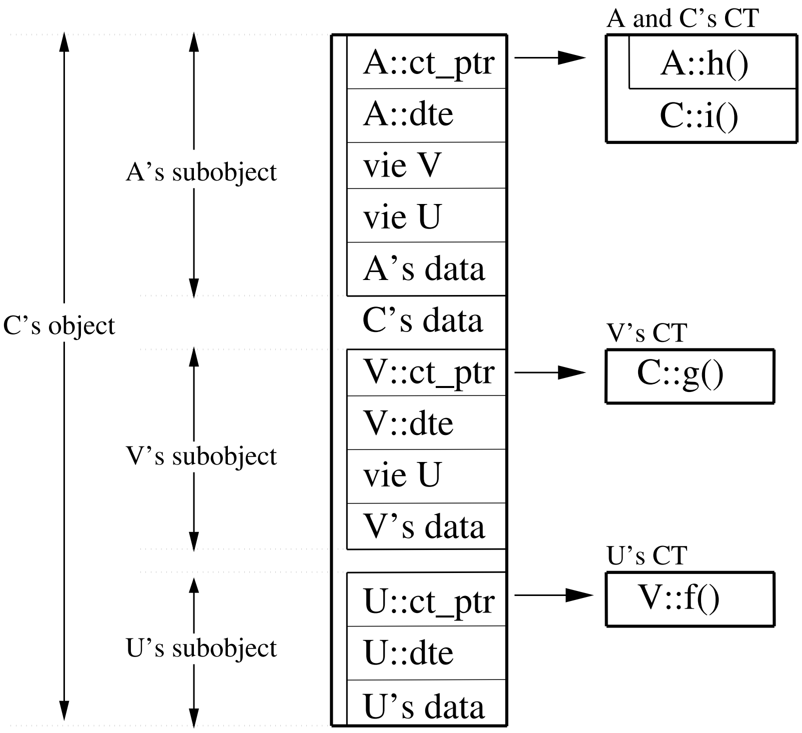

Figure 2: The `ARM` memory layout for class $\mathcal{C}$ when language features are in object.

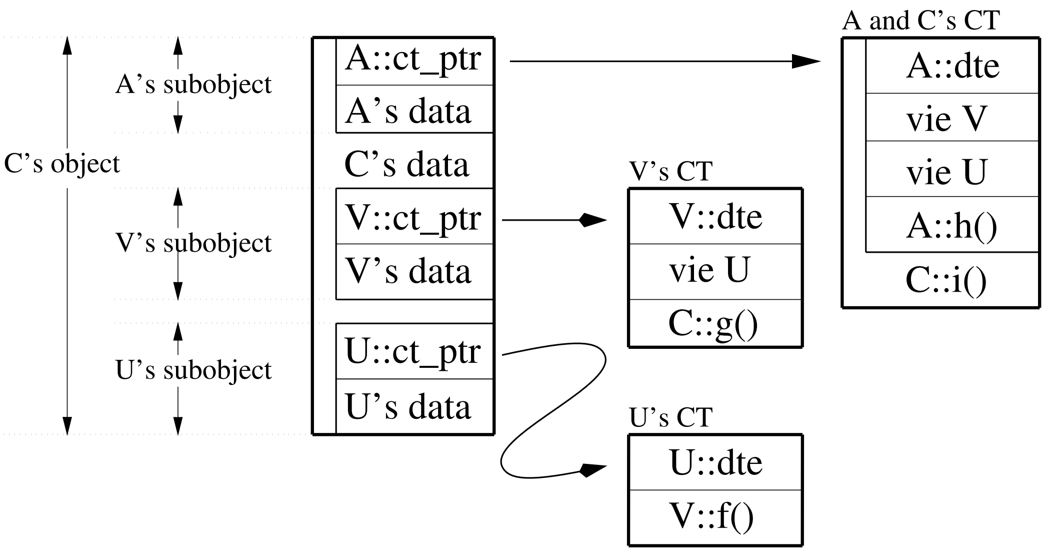

Figure 3: The `ALL` memory layout for class $\mathcal{C}$ when language features are in class.

2 Our Approach

This section presents our assumptions on class and object layouts and introduces some definitions that are needed for subsequent discussion. The assumptions on class and object layouts determine the family of memory layouts under consideration. Class information includes all language feature information that is required but not contained in an object. We make the following assumptions about how class information is laid out:

- class information for a particular class is aggregated into one or more class tables;
- all class tables for a particular class and a particular set of language features have the same layout: that is, the same order and number of entries;
- even if only one entry in a class table changes, a new table is required;

A *class hierarchy graph* [5, 8] (CHG) is a directed acyclic graph that represents a program's class inheritance hierarchy: each node represents a class, and each edge represents a direct inheritance relationship between two classes. Where the meaning is clear from context, we shall refer to a class by the node which represents it, and shall refer to an inheritance relationship by the edge which represents it. If class B inherits from class C, then there is an edge from the B node to the C node. If the inheritance is virtual, the edge is labeled with V. A path from class D to class B in the CHG corresponds to a particular subobject (or instance) of class B that occurs in the D object. Where this path does not include a virtual inheritance edge, B is a nonvirtual subobject of D. Where on this path the incoming edge to B is a virtual inheritance edge, B is a virtual subobject of D.

Objects contain user-defined fields and compiler-generated fields. Compiler-generated fields is the run-time information that is required to implement object-oriented language features. If language feature information is required for class C and is contained in C's class table, then every C object must contain at least one compiler-generated field, a CT_PTR, that points to a class table of C. If a class does not use a particular language feature, then it does not require class or object information for that feature. We use the following definitions by Ellis and Stroustrup [10]: "A *complete object* is an object that is not a sub-object representing a base class. Its class is said to be the *most derived* class for the object." We make the following assumptions about object layout [10]:

- access to data members is fast; that is, given a subobject, its fields can be accessed without indirection;
- the offset between a virtual base subobject and the subobject of a derived class is not fixed with respect to subsequent derivations of that class;
- the offset between a nonvirtual base subobject and the subobject of a derived class is fixed with respect to subsequent derivations of that class.

A *pattern* identifies when language feature information is required by a class or when it changes its value, and hence a new class table is required. Patterns are used to compute the set of class tables required. The patterns considered in this paper will include only those needed for the family of memory layouts defined by the above assumptions. A pattern consists of two input components: path pattern and constraints; and generates two output components: a class table and an entry in the table.

The *path pattern* component is used to identify paths in the CHG. It is specified by a sequence of *class variables* ($\mathcal{D}, \mathcal{C}, \mathcal{B}, \mathcal{U}$, and $\mathcal{V}$) and *path variables* (α, β, and γ), where a class variable assumes classes in the CHG as possible values, and a path variable assumes a possibly empty sequence of classes. Given two class variables $\mathcal{D}$ and $\mathcal{C}$ in a path, we assume that their values are always distinct unless stated otherwise. For the example of Figure 1, $[DBA]$ specifies the path originating at D and terminating at A which traverses B and the path pattern $[\mathcal{D}\alpha\mathcal{A}]$ matches the path $[DBA]$ when the class variable $\mathcal{D}$ is bound to class D, the path variable α is bound to the sequence of classes that consists of only B, and the class variable $\mathcal{A}$ is bound to the class A[1]. In the path pattern $[\mathcal{D}\alpha\mathcal{A}]$, the class that $\mathcal{D}$ is bound to is the most derived class.

We use paths in the CHG to identify subobjects in essentially the same manner as [14] [2]. In the example of Figure 1, two subobjects of type A occur in a D object and are identified by the paths $[DBA]$ and $[DCA]$. Since there is only one virtual base subobject for the multiple paths in the CHG to it from a derived object, these multiple paths may be denoted as a single path when convenient. In the example of Figure 1, V is a virtual base class of D. The set of paths from D to V, $[DBAV]$ and $[DCAV]$, may be denoted as $[DV]$.

The *constraints* component specifies the constraints on nodes and edges in the path that is defined in the path component. For the purposes of this paper, a node constraint specifies a *method variable* ($\mathcal{F}$) that assumes as its value a virtual function defined by a class. An edge constraint specifies that the inheritance between two nodes is virtual. If there is no edge constraint, the path may or may not contain a virtual inheritance edge, unless otherwise stated.

Path patterns are matched against paths in the CHG, where the class variables are bound to classes in the CHG and path variables match any, possibly empty, sequence of classes. When a match is found and all constraints of the constraint component are met, a class table is required for some memory layout in our family of memory layouts. We represent the table as an abstract class table representation (ACTR). An ACTR precisely captures a subobject's context and provides a representation that we symbolically manipulate as discussed in Section 4.2. An ACTR has three components: [3] 1) the most derived class; 2) the path in the CHG corresponding to the subobject whose class table has changed; 3) the path in the CHG corresponding to the subobject whose class table is being constructed or destructed. In the example of Figure 1, the ACTR for the subobject of class V when a subobject of class CA is constructed in a D object would be: $< D, DV, DCA >$. The justification for specifying paths in an ACTR's second and third components is because there could be multiple subobjects of the same type in an object and these subobjects need to be distinguished.

The other output component is the class table entry. This component identifies either a new class entry or a class entry whose value has changed. A class entry's value may change when its subobject context changes due to inheritance, and object construction and destruction.

3 Computing the Space Overhead for Class Tables

For each C++ language feature under consideration, this section presents the patterns that specify the conditions under which a language-feature entry is required or when its value may change. Sections 3.1- 3.3 specify for each language feature in isolation the patterns which determine class table requirements. Where the information for more than one language feature is placed in a class table, we must consider whether any class table is required which is not accounted for when each

[1] The path pattern $[\mathcal{D}\alpha\mathcal{A}]$ also matches the path $[DCA]$ when the path variable α is bound to C.

[2] Initially, we used Rossie and Friedman's subobject labels in our path pattern component and matched path patterns on subobject graphs (an abstract layout of an object); however, we found that the use of their formalism was not well-suited for describing path patterns and added an additional layer of abstraction and therefore complexity into our methodology.

[3] An ACTR could be represented by two components as the first component is redundant; however for expository purposes, we have included the first component in the representation.

language feature is considered in isolation. Section 3.4 specifies these interactions between language features.

There is a total of nine patterns. Four of the patterns are for virtual dispatch; two for virtual inheritance; one for dynamic typing; and two for language interactions: one for dynamic typing and virtual dispatch, and one for dynamic typing and virtual inheritance. The example CHG in Figure 1 is used as an example throughout this section. Each pattern has a table that represents the ACTRs that are generated when that pattern is applied to the CHG in Figure 1. Each row in a table presents the ACTR that is generated, the new entry or the entry whose value has changed, and the mapping of class variables to classes and the mapping of path variables to sequences of classes. When a path variable maps to the empty sequence, this mapping is not specified in the table. If the mapping of a class variable is obvious from the entry, the class variable's mapping is not provided again.

3.1 Virtual Function Class Table Entries

A virtual function entry, VFE, specifies which implementation of a method is to be invoked and what (if any) adjustment to the THIS pointer is to be made so that the correct subobject is pointed to. A virtual function entry may change if either the method implementation or the THIS adjustment changes. We introduce the next three definitions for use in specifying the patterns' constraint components. A virtual function is *introduced* in a class C if that virtual function is defined in C and not defined in any base class of C. A virtual function is *visible* in a class C if the virtual function is introduced either in C or in a base class of C. A virtual function is *overridden* in a class C if the virtual function is defined in C and defined in some base class of C.

The set of relevant patterns is determined by the family of memory layouts under consideration. In addition to the assumptions about class and object layout described in Section 2, we assume a class table contains entries for all virtual functions that are introduced by that class [4]. Also, we assume that VFEs are always contained in class tables [5].

We now present the patterns that specify when a VFE is required or when a VFE's value may change.

Pattern 1: virtual dispatch
Path: $[\mathcal{D}]$
Constraints: $\mathcal{D}$ introduces a virtual function $\mathcal{F}$.
ACTR Generated: $<\mathcal{D}, \mathcal{D}, \mathcal{D}>$
Entry: $\mathcal{D}$::$\mathcal{F}$

Pattern 1 identifies when a new VFE is required for a class. Each pattern specifies a Path and a Constraints component as its inputs and an ACTR Generated and an Entry component as its outputs. In the path component of Pattern 1, $\mathcal{D}$ is a class variable that may be bound to any class. The constraint component introduces the method variable $\mathcal{F}$ that may be bound to any virtual function and states that $\mathcal{F}$ must be bound to a virtual function that is introduced in the class that $\mathcal{D}$ is bound to. Table 1 demonstrates the ACTRs that are generated when Pattern 1 is applied to the CHG of Figure 1. Consider the first row of Table 1. In the first column, the ACTR $< U, U, U >$ has been generated because of the introduction of virtual function f in class U. The second column specifies that the entry $U :: f$ in $< U, U, U >$ is required; that is, $\mathcal{D}$ is bound to U and $\mathcal{F}$ is bound to f. Each class that introduces a virtual function in Figure 1 has a corresponding ACTR in Table 1.

[4] Although there are alternative ways to implement class table entries for virtual dispatch, this approach is the most space efficient. See [23] for more details.

[5] All commercial implementations of C++ known to the authors implement virtual function information this way.

ACTRs	$\mathcal{D} :: \mathcal{F}$
<U, U, U>	U::f
<V, V, V>	V::g
<A, A, A>	A::h
<C, C, C>	C::i

Table 1: The results of applying Pattern 1 to the CHG in Figure 1.

ACTRs	$\mathcal{D} :: \mathcal{F}$	**where**
<V, VU, V>	V::f	$\mathcal{B}$ = U
<B, BU, B>	B::f	$\mathcal{B}$ = U
<A, AV, A>	A::g	$\mathcal{B}$ = V
<C, CV, C>	C::g	$\mathcal{B}$ = V
<D, DV , D>	D::g	$\mathcal{B}$ = V
<D, DBA, D>	D::h	$\mathcal{B}$ = A, $\alpha = \mathcal{B}$
<D, DCA, D>	D::h	$\mathcal{B}$ = A, $\alpha = \mathcal{C}$

Table 2: The results of applying Pattern 2 to the CHG in Figure 1.

Pattern 2: virtual dispatch
Path: [$\mathcal{D}\alpha\mathcal{B}$]
Constraints: 1. $\mathcal{B}$ introduces a virtual function $\mathcal{F}$, and
2. $\mathcal{D}$ overides $\mathcal{F}$.
ACTR Generated: <$\mathcal{D}$, $\mathcal{D}\alpha\mathcal{B}$, $\mathcal{D}$>
Entry: $\mathcal{D}$::$\mathcal{F}$

Pattern 2 identifies when the value of a VFE changes due to method override. In the path component, α is a path variable that may be bound to a possibly empty sequence of classes, and the path [$\mathcal{D}\alpha\mathcal{B}$] requires that $\mathcal{B}$ is a base class of $\mathcal{D}$. The constraint states that the virtual function bound to $\mathcal{F}$ must be introduced in the class bound to $\mathcal{B}$ and overridden in the class bound to $\mathcal{D}$. Table 2 demonstrates the ACTRs that are generated when Pattern 2 is applied to the CHG of Figure 1. Consider the first row of Table 2. The ACTR $< V, VU, V >$ results from the overriding in class V of the virtual function f that is introduced in class U. Since U introduces f, U's class table's entry for f has a different value of $V :: f$. The third column specifies the binding of the class variable $\mathcal{B}$ in the path pattern to class U. The last two rows of Table 2 illustrate an interesting property. The virtual function h is introduced in class A and overridden in class D. In a class D object, the entry for h in A's class table is $D :: h$. But a D object has two subobjects of type A. The THIS adjustment for this entry differs depending on whether the A subobject is contained in a B subobject or a C subobject. Thus two different ACTRs are generated by the overriding of h in D. In this case the path information provided by the second component of the ACTRs, which gives the path from the most derived class to the class that requires a new table, is needed to distinguish the two ACTRs, as seen in the last two entries in Table 2.

ACTRs	$\mathcal{B} :: \mathcal{F}$	**where**
<A, AU, A>	V::f	$\mathcal{D}$ = A, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<C, CU, C>	V::f	$\mathcal{D}$ = C, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<D, DU, D>	B::f	$\mathcal{D}$ = D, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<B, BV, B>	A::g	$\mathcal{D}$ = B, $\mathcal{V}$ = V, $\mathcal{U}$ = V

Table 3: The results of applying Pattern 3 to the CHG in Figure 1.

Pattern 3: virtual dispatch
Path: $[\mathcal{D}\alpha\mathcal{B}\ \mathcal{V}\beta\mathcal{U}]$ where $\mathcal{U}$ may be $\mathcal{V}$ when β is empty
Constraints: 1. $\mathcal{V}$ is a virtual base of $\mathcal{B}$,
2. $\mathcal{U}$ introduces a virtual function $\mathcal{F}$,
3. $\mathcal{B}$ overrides $\mathcal{F}$, and
4. $\mathcal{B}$::$\mathcal{F}$ is not overridden in $\mathcal{D}$ or in a base of $\mathcal{D}$ that is derived from $\mathcal{B}$.
ACTR Generated: $<\mathcal{D}, \mathcal{D}\ \mathcal{V}\beta\mathcal{U}, \mathcal{D}>$
Entry: $\mathcal{B}$::$\mathcal{F}$

Pattern 3 identifies when the value of a VFE's THIS adjustment changes due to a change in the offset from an object to a virtual base subobject. The constraint states that $\mathcal{V}$ is a virtual base of $\mathcal{B}$; that is, there is a path from $\mathcal{B}$ to $\mathcal{V}$ in the CHG such that the incoming edge to $\mathcal{V}$ is labeled with V. In the path pattern, both α and β are path variables. By convention, in our path patterns, all inheritance in a subpath that occurs after virtual inheritance is nonvirtual. In Pattern 3 all inheritance in the subpath $[\mathcal{V}\beta\mathcal{U}]$ is nonvirtual; otherwise, if there exists a $\mathcal{C}$ that is a virtual base of $\mathcal{V}$ in β, then the path pattern would be expressed as $[\mathcal{D}\alpha\mathcal{B}\ \mathcal{C}\gamma\mathcal{U}]$ instead. Table 3 demonstrates the ACTRs that are generated when Pattern 3 is applied to the CHG of Figure 1. Consider the first row of Table 3. The ACTR $< A, AU, A >$ indicates that the entry for f in the class table for U changes when U is a subobject in an A object. In Figure 1, virtual function f is introduced in class U and overridden in class V, where U is a virtual base class of V. In a complete object of class V or A, the THIS adjustment for f is the offset between U and V. In general, this offset differs between a complete class A object and a complete class V object, because the offset of a virtual base is not fixed.

Pattern 4: virtual dispatch
Path: $[\ \mathcal{D}\alpha\mathcal{C}\beta\mathcal{B}\ \mathcal{V}\gamma\mathcal{U}]$ where $\mathcal{D}$ may be $\mathcal{C}$ when α is empty and $\mathcal{U}$ may be $\mathcal{V}$ when γ is empty
Constraints: 1. $\mathcal{V}$ is a virtual base of $\mathcal{B}$,
2. $\mathcal{U}$ introduces a virtual function $\mathcal{F}$,
3. $\mathcal{B}$ overrides $\mathcal{F}$, and
4. $\mathcal{C}$ overrides $\mathcal{F}$.
ACTR Generated: $<\mathcal{D}, \mathcal{D}\ \mathcal{V}\gamma\mathcal{U}, \mathcal{D}\alpha\mathcal{C}\beta\mathcal{B}>$
Entry: $\mathcal{B}$::$\mathcal{F}$

Pattern 4 identifies when a VFE's THIS adjustment changes during object construction and destruction due to the change in the offset from a subobject to a virtual base subobject. The ACTR Generated component identifies that this class table is required during object construction and destruction because the third component of the ACTR specifies the class that $\mathcal{B}$, not $\mathcal{D}$, is bound to. Table 4 demonstrates the ACTRs that are generated when Pattern 4 is applied to the CHG of Figure 1. Consider the first row of Table 4. This row states that the value of the VFE for f changes in the class

ACTRs	$\mathcal{B} :: \mathcal{F}$	**where**
<B, BU, BV>	V::f	$\mathcal{D}$ = B, $\mathcal{C}$ = B, $\mathcal{V} = \mathcal{U}$ = U
<D, DU, DV>	V::f	$\mathcal{D}$ = D, $\mathcal{C}$ = B, $\mathcal{V} = \mathcal{U}$ = U
<C, CV, CA>	A::g	$\mathcal{D}$ = C, $\mathcal{C}$ = C, $\mathcal{V} = \mathcal{U}$ = V
<D, DV, DC>	C::g	$\mathcal{D}$ = D, $\mathcal{C}$ = D, $\mathcal{V} = \mathcal{U}$ = V
<D, DV, DCA>	A::g	$\mathcal{D}$ = D, $\mathcal{C}$ = C, $\mathcal{V} = \mathcal{U}$ = V
<D, DV, DCA>	A::g	$\mathcal{D}$ = D, $\mathcal{C}$ = D, $\mathcal{V} = \mathcal{U}$ = V, β = C
<D, DV, DBA>	A::g	$\mathcal{D}$ = D, $\mathcal{C}$ = D, $\mathcal{V} = \mathcal{U}$ = V, β = B

Table 4: The results of applying Pattern 4 to the CHG in Figure 1.

table for class U when U is a subobject in a B object and the V subobject is being constructed. A new class table is required because the VFE for f in U's class table is $U :: f$ in $< U, U, U >$, $V :: f$ in $< V, VU, V >$, and $B :: f$ in $< B, BU, B >$. In accordance with C++'s semantics for object construction and destruction [1], U's VFE for f when V is being constructed and destructed in a B object is $V :: f$ and the this adjustment is the offset from the U subobject to the V subobject in a B object. The offset between the subobjects of a class and its virtual base is not fixed and in general will differ; therefore, the class table $< V, VU, V >$ for a V object cannot be used for the construction of the V subobject in a B object.

The fifth and sixth entries in Table 4 illustrate that two different path matches by the same pattern against the CHG can result in the same ACTR. Note that the bindings of class variables to classes differ in the two cases. The last entry of Table 4, in comparison with the fifth and sixth entries, also illustrates an interesting property. Since a D object has two subobjects of class A, the THIS adjustment for the $A :: g$ entry in the V class table differs depending on whether the A subobject being constructed is contained in a B subobject or a C subobject. There must be an ACTR for each case. The two ACTRs are distinguished by their differing path entries in the third ACTR component, which gives the path from the most derived class to the class being constructed.

3.2 Virtual Inheritance Entries

A virtual inheritance entry, VIE, provides access to a virtual base subobject of an object. Unlike virtual function entries, a VIE may be stored either in an object as virtual base pointers or offsets, or in a class table as virtual base offsets; and the VIEs for a class are constant during the lifetime of an object. That is, the value of a VIE does not change during subobject construction or destruction.

Pattern 5: virtual inheritance
Path: [$\mathcal{B}$ $\mathcal{V}$]
Constraints: $\mathcal{V}$ is a virtual base of $\mathcal{B}$.
ACTR Generated: <$\mathcal{B}$, $\mathcal{B}$, $\mathcal{B}$>
Entry: $\mathcal{V}$

Pattern 5 identifies when a VIE is needed due to virtual inheritance. The entry specifies the virtual base whose offset is required. Table 5 demonstrates the ACTRs that are generated when Pattern 5 is applied to the CHG of Figure 1. Consider the first row of Table 5. This row states that a new VIE is required for class V. The value of the VIE in $< V, V, V >$ is the offset from V to U in a V object. Due to space considerations, we use the values of $U|V$ for column two in Table 5 to represent the two distinct Pattern 5 matches when $\mathcal{V}$ is bound to U and V, respectively.

ACTRs	$\mathcal{V}$	**where**
<V, V, V>	U	$\mathcal{B}$ = V
<A, A, A>	U | V	$\mathcal{B}$ = A
<B, B, B>	U | V	$\mathcal{B}$ = B
<C, C, C>	U | V	$\mathcal{B}$ = C
<D, D, D>	U | V	$\mathcal{B}$ = D

Table 5: The results of applying Pattern 5 to the CHG in Figure 1.

ACTRs	$\mathcal{V}$	**where**
<A, AV, A>	U	$\mathcal{D}$ = A, $\mathcal{B}$ = V
<B, BV, B>	U	$\mathcal{D}$ = B, $\mathcal{B}$ = V
<C, CV, C>	U	$\mathcal{D}$ = C, $\mathcal{B}$ = V
<D, DV, D>	U	$\mathcal{D}$ = D, $\mathcal{B}$ = V
<B, BA, B>	U|V	$\mathcal{D}$ = B, $\mathcal{B}$ = A
<C, CA, C>	U|V	$\mathcal{D}$ = C, $\mathcal{B}$ = A
<D, DB, D>	U|V	$\mathcal{D}$ = D, $\mathcal{B}$ = B
<D, DC, D>	U|V	$\mathcal{D}$ = D, $\mathcal{B}$ = C
<D, DBA,D>	U|V	$\mathcal{D}$ = D, $\mathcal{B}$ = A, α = B
<D, DCA,D>	U|V	$\mathcal{D}$ = D, $\mathcal{B}$ = A, α = C

Table 6: The results of applying Pattern 6 to the CHG in Figure 1.

Pattern 6: virtual inheritance
Path: [$\mathcal{D}\alpha\mathcal{B}\ \mathcal{V}$]
Constraints: $\mathcal{V}$ is a virtual base of $\mathcal{B}$.
ACTR Generated: $<\mathcal{D}, \mathcal{D}\alpha\mathcal{B}, \mathcal{D}>$
Entry: $\mathcal{V}$

Pattern 6 identifies when a VIE's value changes due to class derivation. Table 6 demonstrates the ACTRs that are generated when Pattern 6 is applied to the CHG of Figure 1. Consider the first row of Table 6. This row states that the value of the VIE for U in V's class table changes when V is a subobject of an A object. In general, the offset between U and V differs in a fully constructed A object from a fully constructed V object, since the offset between a virtual base and the subobject of a derived class is not fixed.

3.3 Dynamic Type Entries

C++ semantics state that the dynamic type entry, DTE, for a fully constructed object specifies the type of the fully derived object; whereas for a partially constructed or destructed object, the DTE specifies the type of the subobject that is being constructed or destructed [1]. Nevertheless, when considering DTEs in isolation, each class requires at most one DTE since that DTE may be used when the class is either fully or only partially constructed. Dynamic type information may be stored either in objects or in class information.

ACTRs	$\mathcal{D}$
<U, U, U>	U
<V, V, V>	V
<A, A, A>	A
<B, B, B>	B
<C, C, C>	C
<D, D, D>	D

Table 7: The results of applying Pattern 7 to the CHG in Figure 1.

Pattern 7: dynamic typing
Path: [$\mathcal{D}$]
Constraints: A virtual function is visible in $\mathcal{D}$.
ACTR Generated: $<\mathcal{D}, \mathcal{D}, \mathcal{D}>$
Entry: $\mathcal{D}$

Pattern 7 identifies when a DTE is required for a class. The constraint states that a class must have a virtual function visible to require a DTE. Table 7 demonstrates the ACTRs that are generated when Pattern 7 is applied to the CHG of Figure 1. Consider the first row of Table 7. When a subobject of class U is constructed, either fully or partially, its class table has a DTE whose value is this same class.

3.4 Language Feature Interactions

This section presents patterns that are required to specify class tables that arise from the interaction between the class table entries for different language features. When class table entries for dynamic typing interact with class table entries for either virtual functions or virtual inheritance, additional class tables may be required.

3.4.1 The Interaction between Dynamic Typing and Dynamic Dispatch

The interaction between the class table entries for virtual functions and dynamic typing can result in additional construction tables for a class.

Pattern 8: virtual dispatch and dynamic typing
Path: [$\mathcal{D}\alpha\mathcal{B}\ \mathcal{V}\beta\mathcal{U}$] where $\mathcal{U}$ may be $\mathcal{V}$ when β is empty
Constraints: 1. $\mathcal{V}$ is a virtual base of $\mathcal{B}$,
2. $\mathcal{U}$ introduces a virtual function $\mathcal{F}$, and
3. $\mathcal{B}$ overrides $\mathcal{F}$.
ACTR Generated: $< \mathcal{D}, \mathcal{D}\ \mathcal{V}\beta\mathcal{U}, \mathcal{D}\alpha\mathcal{B}>$
Entry: $\mathcal{B}$ (DTE), and $\mathcal{B}$:: $\mathcal{F}$ (VFE)

When class tables have dynamic typing and virtual function entries, Pattern 8 identifies when a construction class table is required due to a change in a VFE's THIS adjustment or to a change in a DTE. Table 8 demonstrates the ACTRs that are generated when Pattern 8 is applied to the CHG of Figure 1. Consider the first row in Table 8. This row states that a new class table is required for the U subobject in an A object when the V subobject is being constructed. For the ACTR

ACTRs	$\mathcal{B}, \mathcal{B} :: \mathcal{F}$	where
<A, AU, AV>	V, V::f	$\mathcal{D}$ = A, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<B, BU, BV>	V, V::f	$\mathcal{D}$ = B, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<C, CU, CV>	V, V::f	$\mathcal{D}$ = C, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<D, DU, DV>	V, V::f	$\mathcal{D}$ = D, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<D, DU, DB>	B, B::f	$\mathcal{D}$ = D, $\mathcal{V}$ = U, $\mathcal{U}$ = U
<B, BV, BA>	A, A::g	$\mathcal{D}$ = B, $\mathcal{V}$ = V, $\mathcal{U}$ = V
<C, CV, CA>	A, A::g	$\mathcal{D}$ = C, $\mathcal{V}$ = V, $\mathcal{U}$ = V
<D, DV, DC>	C, C::g	$\mathcal{D}$ = D, $\mathcal{V}$ = V, $\mathcal{U}$ = V
<D, DV, DBA>	A, A::g	$\mathcal{D}$ = D, $\mathcal{V}$ = V, $\mathcal{U}$ = V, α = B
<D, DV, DCA>	A, A::g	$\mathcal{D}$ = D, $\mathcal{V}$ = V, $\mathcal{U}$ = V, α = C

Table 8: The results of applying Pattern 8 to the CHG in Figure 1.

$< A, AU, AV >$, the DTE is V and the VFE is $V :: f$. Although the values of the DTE for V in the ACTRs $< V, VU, V >$ and $< A, AU, AV >$ are the same, the values of f's VFE are not the same in these two ACTRs since the offset between V and its virtual base U is different in a V object than in an A object. The value of f's VFE in $< A, AU, AV >$ is the same as in $< A, A, A >$; however, their DTE values differ since $< A, A, A >$ is A. Notice that the third component of the ACTRs, which gives the path from the most derived class to the class being constructed, is needed to distinguish among the last three ACTRs in Table 8.

Although this section discusses the interaction between virtual function and dynamic type entries, this pattern requires the existence of virtual inheritance, but does not make any requirement on virtual inheritance entries.

3.4.2 The Interaction between Dynamic Typing and Virtual Inheritance

The interaction between class table entries for virtual inheritance and dynamic typing can result in additional construction tables for a class.

Pattern 9: virtual inheritance and dynamic typing
Path: $[\mathcal{D}\alpha\mathcal{B}\ \mathcal{V}]$
Constraints: 1. $\mathcal{V}$ is a virtual base of $\mathcal{B}$, and
2. a virtual function is visible in $\mathcal{B}$.
ACTR Generated: $<\mathcal{D}, \mathcal{D}\alpha\mathcal{B}, \mathcal{D}\alpha\mathcal{B}>$
Entry: $\mathcal{B}$ (DTE), and $\mathcal{V}$ (VIE)

When class tables contain both dynamic typing and virtual inheritance entries, Pattern 9 identifies when a new class table is required due to a change in a VIE or a change in a DTE. Table 9 demonstrates the ACTRs that are generated when Pattern 9 is applied to the CHG of Figure 1. Consider the first row in Table 9. This row states that a new class table is required for the V subobject in an A object when the V subobject is being constructed. For $< A, AV, AV >$ the DTE is V and the VIE is the offset from V to U in an A object. Although the values of the DTEs for V in the ACTRs $< A, AV, AV >$ and $< V, V, V >$ are the same, the values of U's (or V's) VIE in these two ACTRs is not the same since the offset between a subobject of a class and its virtual base is not fixed. The value of U's (or V's) VIE in $< A, AV, AV >$ is the same as in $< A, A, A >$; however, their DTE

ACTRs	$\mathcal{B}, \mathcal{V}$	**where**
<A, AV, AV>	V, U	$\mathcal{D}$ = A
<B, BV, BV>	V, U	$\mathcal{D}$ = B
<C, CV, CV>	V, U	$\mathcal{D}$ = C
<D, DV, DV>	V, U	$\mathcal{D}$ = D
<B, BA, BA>	A, V|U	$\mathcal{D}$ = B
<C, CA, CA>	A, V|U	$\mathcal{D}$ = C
<D, DB, DB>	B, V|U	$\mathcal{D}$ = D
<D, DC, DC>	C, V|U	$\mathcal{D}$ = D
<D, DBA, DBA>	A, V|U	$\mathcal{D}$ = D, α = B
<D, DCA, DCA>	A, V|U	$\mathcal{D}$ = D, α = C

Table 9: The results of applying Pattern 9 to the CHG in Figure 1.

values differ since $< A, A, A >$'s value is A. Notice that in Table 9 the class being constructed is the same as the class whose table is changed; and therefore, the second and third components of each ACTR are identical.

In that the VIEs are constant, they might profitably be placed in a virtual inheritance class table, VIT, that is separate from a virtual function class table. In this case, each object whose class has a virtual base would require, in addition to perhaps a CT_PTR, the pointer to the virtual function table, a VIT_PTR, the pointer to a VIT of that class. One of the memory layouts studied in Section 4 uses this technique. [6]

We conclude this section with some general observations about patterns. Six of the nine patterns occur only in the presence of virtual inheritance: virtual inheritance introduces significant complexity for class table generation. Of those six Patterns, three (4, 8, and 9) generate construction class tables, and the interaction between language features accounts for two of these three patterns. An additional conclusion is that C++'s construction semantics and the interaction between language features also contributes to the complexity of class table generation.

4 Experimental Results

Given a set of five benchmarks, this section compares the run-time space efficiency of alternative memory layouts, each of which represents a set of choices for the implementation of virtual dispatch, virtual inheritance, and dynamic type casting. In particular, this section answers the following questions about the space overhead for these object-oriented language features:

1. What is the space overhead?
2. What proportion of the space overhead is due to class tables?
3. How much does "sharing" affect space overhead?
4. What percentage of class table overhead is due to language feature interaction?

[6] Placing the DTEs and VFEs in one table and VIEs in a separate table eliminates additional class tables that arise due to the interaction between dynamic typing and virtual inheritance, although additional tables may still be required due to the interactions between dynamic typing and virtual dispatch.

Answering the above questions for each memory layout and our set of benchmarks provides a quantitative comparison among memory layouts with respect to space overhead requirements. In what follows, *overhead*, unless specified otherwise, always refers to space overhead.

Section 4.1 describes each memory layout that we consider. Section 4.2 describes the measurement methodology that is used. Section 4.3 gives a brief overview of the benchmarks. Section 4.4 presents the data which can be used to answer the above questions.

4.1 Memory Layouts

The five memory layouts are: ARM, VF_VI, IBM, MS, and ALL. ARM denotes the standard C++ implementation [10]. As illustrated in Figure 2, this memory layout places VFEs in class tables; and places VIEs, DTEs, and pointers to class tables in objects. Patterns 1–4 are needed to compute the class tables that are required by ARM. VF_VI denotes the memory layout that places VFEs and VIEs in class tables, and places pointers to class tables and DTEs in objects. Patterns 1–4, 5, and 6 are needed to compute the class tables that are required by VF_VI. Both ARM and VF_VI have no class tables due to the interaction between language features. IBM denotes the memory layout which places VFEs and DTEs in class tables, and places pointers to class tables and VIEs in objects. This memory layout closely resembles IBM's Visual Age C++ compiler [5]. Patterns 1–4, 7, and 8 are needed to compute its class tables. MS denotes the memory layout that places class pointers in objects; and places VFEs, VIEs and DTEs in class tables. However, the VIEs are partitioned into a separate type of table, denoted a virtual inheritance table. Any object whose class has a virtual base requires a pointer to a virtual inheritance table. This memory layout is one feature of the Microsoft's Visual C++ compiler [15]. Patterns 1–8 are needed to compute the class tables that are required by MS. Patterns 5 and 6 are used to generate the virtual inheritance tables, and Patterns 1–4, 7, and 8 are used to generate the other tables. By placing VIEs and DTEs in separate tables, there is no interaction between these language features, and therefore, MS does not require Pattern 9. As illustrated in Figure 3, ALL denotes the memory layout that places VFEs, VIEs and DTEs in class tables and places class table pointers in objects. Patterns 1–9 are needed to compute the tables required by ALL. The ARM and the ALL memory layouts represent two extremes of the spectrum: ARM minimizes class entries and maximizes object entries, and ALL minimizes object entries and maximizes class entries. Comparing the class table overhead of ARM and VF_VI for those benchmarks that have VIEs determines the class table overhead for having VIEs as class entries. In addition, the difference in the class table overhead of MS and ALL for those benchmarks that have VIEs is due solely to the interaction between dynamic typing and virtual inheritance. Finally, benchmarks that have no virtual inheritance need only Patterns 1, 2, and 7 to compute their class tables.

4.2 Experimental Methodology

A class hierarchy graph is built from the inheritance and method declarations of each class. This information was obtained by instrumenting a C++ compiler developed at IBM [16]. The patterns are applied to an application's CHG to compute the ACTRs required by each class. The set of ACTRs that are computed is independent of a particular memory layout. For a particular class, its ACTRs determine how many entries are required for each language feature.

The memory layout determines where the language feature information is stored. Computing the overhead for a particular memory layout, $\mathcal{M}$, requires computing the overhead due to objects and due to class tables. The table overhead for each class $\mathcal{C}$ with respect to $\mathcal{M}$ is computed as the sum of the tables "required" for $\mathcal{C}$ by $\mathcal{M}$ times the number of entries required for a C class table with respect to $\mathcal{M}$. The tables that are required for $\mathcal{C}$ by $\mathcal{M}$ is the number of distinct ACTRs for $\mathcal{C}$ that are generated by patterns whose language features are stored in class tables by $\mathcal{M}$.

Consider class C in Figure 1. For the ARM memory layout, class C requires one entry in one class table: $< C, C, C >$ generated by Pattern 1. For the ALL memory layout, class C requires four entries in two class tables. The class tables are $< C, C, C >$ generated by Patterns 1, 5, and 7; and $< D, DC, D >$ generated by Patterns 6 and 9. The breakdown of the four entries is two VIEs, one VFE, and one DTE.

The object overhead for each class $\mathcal{C}$ with respect to a memory layout, $\mathcal{M}$, is computed as the number of objects created for $\mathcal{C}$ times the number of object entries required for $\mathcal{C}$ with respect to $\mathcal{M}$. An instruction trace of an application's execution was used to simulate the creation of objects to obtain the number of objects each class instantiates.

A standard technique for reducing space overhead is sharing language feature information between subobjects and class tables [18, 10]. A full explanation of sharing is outside the scope of this paper; however, we discuss how sharing works for virtual dispatch and how we model this sharing by symbolically manipulating ACTRs. Other language feature information can be modeled similarly. All of the patterns presented in Section 3 assume that no sharing occurs.

The class table pointers that are required by virtual dispatch may be shared between a subobject and one of its direct, nonvirtual base class subobjects. The sharing of class tables pointers is not limited to two subobjects but may chain together a sequence of subobjects whose classes form a path in the class hierarchy graph that does not contain any virtual inheritance. If an application only uses single and nonvirtual inheritance, then sharing ensures that every object will contain at most one class table pointer. When the subobjects of two classes share class table pointers, the class tables must be combined by aggregating all their entries together. This aggregation could increase the overall class table size since a change to an entry in either class requires a new aggregated class table to be generated. Alternatively, the number of entries in the tables of classes that share may decrease due to sharing. That is, if class $\mathcal{C}$ and class $\mathcal{A}$ share a class table, every entry that occurs in both class tables only occurs once in the aggregated class table.

With multiple inheritance, a class may have multiple direct, nonvirtual base classes with which it can share class tables. Which base class should be chosen? For purposes of this paper, we assume that a class shares with its first (in declaration order) direct, nonvirtual base class

Consider again class C from Figure 1. In both Figures 2 and 3, class C shares with its base class A. Whenever a C subobject exists, it shares a CT_PTR with its A subobject and the CT_PTR points to a class table with both A and C entries. For example, in both Figures 2 and 3 there is one class table that is shared by the C object and the A subobject. Further, a C subobject can reuse its A subobject's language feature entries. In addition to sharing CT_PTRs, Figures 2 illustrates that for ARM the DTE and VIEs are shared between the C object and the A subobject.

We model sharing by symbolically manipulating ACTRs. If class $\mathcal{C}$ and class $\mathcal{A}$ share, then the entries of an $\mathcal{A}$ ACTR that are not replicated in a $\mathcal{C}$ ACTR are added to $\mathcal{C}$'s ACTR. Whenever both an $\mathcal{A}$ and a $\mathcal{C}$ ACTR occur in the same context, the $\mathcal{C}$ ACTR obtains the nonreplicated entries of the $\mathcal{A}$ ACTR and $\mathcal{A}$'s ACTR is eliminated.

Since the entries for language features are disjoint, we can examine the ACTRs for each language feature in turn to determine the effect of sharing. Consider Figure 1 when A and C share. The ACTRs of both C and A are affected by sharing as follows. A new VFE for h is added to all C ACTRs. In particular, it is added to $< C, C, C >$. No new VIEs are added to C ACTRs since all of A's VIEs are replicated in C's ACTRs. The ACTRs $< C, CA, C >$ and $< D, DCA, D >$, which are generated by Pattern 6, are eliminated, since they occur in the same context as $< C, C, C >$ and $< D, DC, D >$, respectively. No new DTEs are added to C ACTRs since such an entry already occurs in every C ACTR. The interaction between language features does not effect C or A's ACTRs. In particular,

Programs	Description	LOC	No. of Classes		Objects Created
			Total	Virtual	
`idl`	SunSoft IDL compiler + demo back end	30,288	87	26	26,971
`npic`	Framework for interprocedural analysis	22,728	198	42	2,221,492
`jikes`	IBM Java to byte code translator	58,296	270	0	15,323
`lcom`	”L” hardware description Language compiler	17,278	78	0	147,195
`ixx`	IDL specification to C++ code generator	11,157	103	0	38,721

Table 10: The set of benchmark programs used to evaluate the space overhead of different object models.

both the $< D, DC, DC >$ and $< D, DCA, DCA >$ ACTRs, which are generated by Pattern 9, must be retained since the subobjects that are being constructed differ.

4.3 Benchmark Characteristics

Table 10 shows the five benchmarks that were used to evaluate the overhead of different memory layouts. The columns of the table show for each benchmark: the name of the application, a short description, the size (number of lines of source code), the number of classes in the application's class hierarchy, the total number of virtual bases, and the total number of objects that were created for a particular execution of the program. The library classes for these applications do not use any object-oriented features and so are excluded from the count of classes.

Two benchmarks are extensible frameworks which make heavy use of virtual inheritance. `npic` (New Paltz Interprocedural Compiler) is a program analysis system which defines a class framework for interprocedural analysis. The benchmark implements flow-sensitive and flow-insensitive algorithms for pointer alias analysis [13]. `idl` is the front end of a compiler for interface definition language specifications. It is designed so that back ends may be easily added through class derivation. The other three benchmarks, `jikes`, `lcom`, and `ixx`, do not utilize multiple or virtual inheritance. They use single inheritance only.

4.4 Measurements

This section presents the measurements that we have obtained and will be used to answer the four questions presented at the beginning of Section 4.

Table 11 presents the object-oriented overhead for our set of benchmarks with respect to the `ARM` and `ALL` memory layouts. That is, it addresses our first question: What is the overhead for object-oriented features? Object-oriented overhead includes the size of class tables and of the compiler-generated fields required in objects to implement object-oriented language features. The column labeled ”user-defined fields” is the aggregate overhead (in bytes) required to represent objects without any object-oriented overhead; that is, without compiler-generated fields or class tables. The columns labeled ” `ARM`” and ” `ALL`” identify the respective memory layouts. For a benchmark, corresponding to a row, and a memory layout, corresponding to a column, each entry contains the object-oriented overhead (also in bytes) and its percentage with respect to the user-defined fields.

Table 11 indicates a large variation between benchmarks in object-oriented overhead. For `ARM`, the overhead ranges from 99.79%, almost matching the overhead for the user-defined fields for `idl`, to 1.45% for `npic`. The `ARM` memory layout increases object-oriented overhead at least 95%

Benchmarks	user-defined fields	Object-Oriented Overhead	
		ARM	ALL
idl	395,316	394,496 (99.79%)	141,852 (35.88%)
npic	25,152,112	365,116 (01.45%)	104,880 (00.42%)
jikes	302,760	95,140 (31.42%)	48,752 (16.10%)
lcom	2,793,984	142,520 (05.10%)	72,976 (02.61%)
ixx	435,918	264,184 (60.60%)	134,416 (30.84%)

Table 11: The object-oriented overhead for our set of benchmarks with respect to the **ARM** and **ALL** memory layouts.

Benchmarks	Class Table Overhead			
	ARM		ALL	
	Total	Tables	Total	Tables
idl	394,496	8,076 (2.05%)	141,852	21,604 (15.23%)
npic	365,116	17,060 (4.67%)	104,880	38,476 (36.69%)
jikes	95,140	1,476 (1.55%)	48,752	1,920 (03.94%)
lcom	142,520	3,032 (2.13%)	72,976	3,232 (04.43%)
ixx	264,184	4,328 (1.64%)	134,416	4,528 (03.37%)

Table 12: The class table and object overhead for the set of benchmarks with respect to the `ARM` and `ALL` memory layouts.

Memory Layouts	Object-Oriented Overhead			
	idl		npic	
	Total	Tables	Total	Tables
ALL	141,852(35.88%)	21,604(15.23%)	104,880 (0.42%)	38,476(36.69%)
MS	182,124(46.07%)	15,708(08.62%)	184,028 (0.73%)	30,456(16.55%)
IBM	246,356(62.32%)	13,984(05.68%)	255,020 (1.01%)	28,716(11.26%)
VF VI	284,436(71.95%)	10,140(03.56%)	207,444 (0.82%)	19,288(09.30%)
ARM	394,496(99.79%)	8,076(02.05%)	365,116 (1.45%)	17,060(04.67%)

Table 13: The object-oriented overhead for the set of benchmarks with respect to all the our memory layout.

Benchmarks	Overhead for Not Sharing			
	Sharing		No Sharing	
	ALL	ARM	ALL	ARM
idl	141,852	394,496	186,872(31.74%)	494,096(25.25%)
npic	104,880	365,116	126,428(20.55%)	408,024(11.75%)
jikes	48,752	95,140	49,176(00.87%)	121,972(28.20%)
lcom	72,976	142,520	115,408(58.15%)	262,804(84.40%)
ixx	134,416	264,184	150,320(11.83%)	343,812(30.14%)

Table 14: The object overhead for the set of benchmarks with sharing and not sharing are considered with the **ARM** and **ALL** memory layout.

over the `ALL` memory layout for all benchmarks. Since `ALL` aggressively moves language feature information into class tables, this technique can have a significant impact on overhead. Further, the two benchmarks, `idl` and `npic`, that heavily use virtual inheritance have the largest variation within the memory layouts `ARM` and `ALL`. As will become explicit with Table 15, an execution of `npic` instantiates few objects of classes that use object-oriented features. Surprisingly, the object-oriented overhead is significant for two out of the three benchmarks that only use single, nonvirtual inheritance. Even with the `ALL` memory layout, `jikes` has 16% and `ixx` has 30% overhead for object-oriented features.

Table 12 presents the proportion of object-oriented overhead that is due to class tables for our set of benchmarks with respect to the `ARM` and `ALL` memory layouts. The table addresses our second question: What is the overhead for class tables? Given a benchmark, corresponding to a row, and a memory layout, corresponding to a column, the "Total" entry contains the total aggregate object-oriented overhead, and the "Tables" entry contains the overhead (in bytes) for class tables and the percentage of class table overhead with respect to the total object-oriented overhead. The entries in the "Total" columns correspond to the entries in the "ARM" and "ALL" columns of Table 11.

The class table overhead for single, nonvirtual inheritance benchmarks is less than 5%, independent of the memory layout. The class table overhead of any benchmark with respect to the `ARM` memory layout is never more than 5%. The only significant class table overhead occurs for benchmarks that have multiple and virtual inheritance with respect to the `ALL` memory layout.

For `idl` and `npic` the variation in class table overhead is pronounced between the memory layouts, increasing from `ALL` to `ARM`. Further, the variation in percentage of total overhead required by class tables is even more dramatic, differing more than a factor of seven, increasing from `ARM` to `ALL`. This phenomenon is due to the reduction in object overhead for the `ALL` memory layout.

Although `npic` has the highest class table overhead, with 36% for `ALL`, `npic`'s object-oriented overhead is less than 1% of its total memory consumption for `ALL`, as illustrated in Table 11. Therefore the impact of class table overhead on the `npic`'s total memory consumption is insignificant. Conversely, `idl`'s class table overhead is approximately 5% of its total memory consumption, 15% of 35%.

In conclusion, the class table overhead for benchmarks without virtual inheritance is an insignificant percentage of the object-oriented overhead for the memory layouts that we are examining. Contrary to popular belief, for benchmarks that use virtual inheritance, the class table overhead should not be a primary factor for choosing one of the memory layouts that we are examining [7].

[7] Microsoft's memory layout [15] trades off object space for class space by introducing a word of storage before each

Alternative memory layouts that significantly reduce object overhead [12] may result in a different conclusion about class table overhead.

Benchmarks that have only single, nonvirtual inheritance have no VIEs. In these benchmarks, the memory layout of VF_VI is equivalent to the memory layout of ARM since DTEs are located in object, and the memory layouts for IBM and MS are equivalent to the memory layout of ALL since DTEs are located in class tables. Table 13 presents all five object models for the only two benchmarks that have virtual inheritance, idl and npic. This table presents the data from Tables 11 and 12 for idl and npic with respect to all the memory layouts. Given a benchmark, corresponding to a column, and a memory layout, corresponding to a row, the "Total" entry contains the aggregate object-oriented overhead and its percentage with respect to the user-defined fields, and the "Tables" entry contains the overhead for class tables and the percentage of class table overhead with respect to the total object-oriented overhead.

From the entries, it is clear that ARM and ALL are the end points of a spectrum. Further, the increase in class table overhead decreases the object-oriented overhead. This confirms the commonly held belief that object overhead dominates class table overhead; since, in general, we expect there to be many instances of each class. The relationship of the overhead between the two memory layouts, IBM and VF_VI, differs for the two benchmarks. For idl, IBM's total overhead is less than VF_VI and for npic it is the reverse, even though the class table overhead is always less for VF_VI than IBM. Recall that the distinction between the two memory layouts is that IBM places DTEs in class tables and VIEs in object, and VF_VI does the reverse. As will be seen in Table 16, with idl the number of VIEs that are in objects with the IBM memory layout is less than the number of DTEs that are in objects with the VF_VI memory layout. The opposite is the case with npic. As one might expect, the overhead for VIEs and DTEs is benchmark dependent.

Consider our fourth question. As noted in Section 4.1, the difference in the class table overhead between MS and ALL is the overhead that is generated by the interaction between dynamic typing and virtual inheritance. In particular, for idl the class table overhead for MS is increased by 37.5% or 5,896 bytes compared to ALL, while npic is increased by 26.3% or 8,020 bytes. The overhead due to the interaction between virtual inheritance and dynamic typing appears to be significant, where virtual inheritance is used. We do not compute the impact of the interaction between virtual dispatch and dynamic typing.

Table 14 illustrates the benefits of sharing and addresses our third question: How much does sharing affect overhead? Sharing is described in Section 4.2. Given a benchmark, corresponding to a row, and a memory layout, corresponding to a column, an entry denotes the object-oriented overhead and a percentage in a "No Sharing" column denotes the increase in object-oriented overhead due to not sharing. Until now, sharing has been assumed, and all results reflect this assumption.

In this table, there is a significant variation between benchmarks, from less than 11.75% (npic) to 84.4% (lcom); and between memory layouts for a benchmark, from less than 1% to more than 28% for jikes. Both memory layouts have benchmarks with high increase in object-oriented overhead when not sharing, 58% for ALL and 84% for ARM, indicating that significant savings may arise from sharing entries both in objects and in class tables. Sharing will be assumed, unless stated otherwise, for all further results.

Table 15 helps explain the wide variation in object-oriented overhead for different benchmarks. This table presents the static and dynamic percentage of objects that contain compiler-generated fields. The static percentage does not take into account an application's execution, and simply considers every class as instantiated once. The dynamic percentage uses the number of times a class is instantiated from an execution of the application to weight the percentage. There is no

virtual base that contains the virtual base's offset to eliminate construction virtual function tables.

% Containing Compiler-Generated Fields		
Benchmarks	**Static**	**Dynamic**
idl	86.2%	100.0%
npic	53.0%	00.3%
jikes	41.1%	76.4%
lcom	64.1%	11.8%
ixx	48.5%	83.9%

Table 15: The static and dynamic percentage of objects that contain compiler-generated fields with sharing.

distinction between subobjects; therefore, if any subobject in an object contains compiler-generated fields, the object is considered to contain compiler-generated fields. The percentages shown here are for the ARM memory layout; only npic varies, and only slightly, when other memory layouts are considered.

Consider the dynamic measurements. npic has the fewest instantiated objects that contain compiler-generated fields, less than 1%; while for idl, every instantiated object has at least one compiler-generated field. This reflects the results presented in Table 11. There is a direct correlation between object-oriented overhead and the percentage of instantiated objects that have compiler-generated fields.

Although a significant number of classes in npic use object-oriented features like virtual inheritance, few if any of these classes are instantiated at run-time. npic is a framework for alias analysis. Although the classes that model program constructs are highly object-oriented, other classes which manipulate the alias sets do not use object-oriented language features. When computing the aliases for a program, the number of alias sets far exceeds the number of program constructs [8].

Another observation is that static measurements are not a good indication of dynamic behavior. From the static measurements, it appears that npic would have significant object-oriented overhead. Other work [25] has confirmed these observations with respect to studying dead data members in C++ applications.

Finally, Table 16 illustrates how compiler-generated fields in idl objects are distributed between language features with respect to each memory layout. The compiler-generated field overhead does not include class table overhead. This overhead is expressed as a percentage of the overhead of user-defined fields. Again, this table illustrates that ARM and ALL are two end points of a spectrum. The entry for MS and VIE, 11.7%, is computed from the number of pointers to class tables that contain VIEs. Recall that MS has separate class tables for VIEs. Interestingly, the overhead required for pointers to virtual inheritance class tables is less than the overhead required for VIEs in object, 28.4 %. We can compute from these two percentages that the average number of VIEs per subobject is 2.43, since there is exactly one pointer wherever a VIE would occur.

[8] Michael Hind, the primary author of this program, confirmed this behavior.

I D L				
Memory Layouts	**Compiler-Generated Fields(%)**			
	Total	**VFE**	**VIE**	**DTE**
1: **ARM** (no sharing)	123.0	41.8	28.4	52.8
2: **ALL** (no sharing)	41.8	41.8	00.0	00.0
3: **ARM**	97.7	30.4	28.4	39.0
4: **VF VI**	69.4	30.4	00.0	39.0
5: **IBM**	58.8	30.4	28.4	00.0
6: **MS**	42.1	30.4	11.7	00.0
7: **ALL**	30.4	30.4	00.0	00.0

Table 16: How object overhead is broken down into compiler-generated fields.

5 Future Work

This paper only considers the space overhead of object-oriented language features. Although time overhead is an important issue, it is not as straightforward to quantify. Consider the `ALL` memory layout, which aggressively moves language feature information into class tables. On the one hand, accessing this language feature information is more costly, as a level of indirection is required. On the other hand, the time overhead to maintain this information at object construction and destruction time is lessened. Only class table pointers needs to be updated, but not DTEs since DTEs are implicitly updated when the class table pointer is updated. To evaluate the time overhead, the execution behavior of an application needs to be characterized in terms of its access to object-oriented information. We have some preliminary information that accessing object-oriented information may not be that frequent. Even if the accesses are frequent, we believe that new optimizations can be developed to reduce the frequency. Nevertheless, a experimental study on the time overhead of object-oriented features is required to resolve these issues.

Currently, our approach does not utilize any form of program analysis of method bodies to reduce space overhead. Analysis of method bodies could allow us to remove dead language feature entries by determining if a particular language feature for a particular class is ever used in an application. Previous work [25] explored this question only for user-defined fields and found promising results.

There are broader families of memory layouts that could be considered by creating new sets of patterns. For example, selective replication of virtual function entries can eliminate accessing a virtual base's class table [5], resulting in faster virtual function dispatch in some cases. Another example is exploring alternative sharing strategies. For example, under certain conditions, a class can share a class table with a virtual base [5]. One could extend the set of patterns described in this paper to measure the tradeoff in space for these alternative memory layouts.

There are several open questions worth exploring. Although we experimentally verified, with different C++ implementations, that our patterns are complete with respect to the family of memory layouts studied here, we would like to prove that our set of patterns is necessary and sufficient to compute the set of ACTRs required by this family.

6 Conclusions

This paper develops a formalism that precisely characterizes when class tables are required for a family of memory layouts. We use this formalism and other techniques to quantify and evaluate, on a set of benchmarks, the space overhead for a set of C++ memory layouts. In particular, this paper studies the space overhead due to three language features: virtual dispatch, virtual inheritance, and dynamic typing. To date, there has been no scientific quantification or evaluation of C++ memory layouts. This paper is an attempt to develop a scientific basis for quantifying and evaluating alternative C++ memory layouts. This work has already influenced the memory layout design choices in the next version of IBM's Visual Age C++ compiler.

Object-oriented space overhead can be dramatic. For one benchmark, the space overhead was almost doubled. The impact can vary dramatically between applications (ranging from 0.42% to 99.79% for our benchmarks). In particular, applications whose object space is dominated by objects that heavily use object-oriented language features will be significantly impacted by the choice of a memory layout. The two end points in the spectrum of our memory layouts differ by at least 95% for each of our benchmarks.

Surprisingly, the object-oriented overhead may be significant for applications that only use single, nonvirtual inheritance. In particular, `jikes`' overhead ranges from 16.1% to 31.42% and `ixx`'s overhead ranges from 30.84% to 60.60%.

The class table overhead for our benchmarks and our set of memory layouts was insignificant. Therefore, at least for our benchmarks and our memory layouts, trying to reduce overall space overhead by minimizing class table overhead is a poor design choice. Alternative memory layouts that significantly reduce object overhead [12] may result in a different conclusion.

Not sharing of language feature information significantly increases object-oriented overhead. Although there is significant variation in the increase, all of our memory layouts have benchmarks with high increase when not sharing, up to 58% for `ALL` and 84% for `ARM`, indicating that significant savings may arise from sharing entries both in objects and in class tables.

The overhead due to the interaction between virtual inheritance and dynamic typing appears to have a significant impact on class table overhead, where virtual inheritance is used. In particular, for `idl` the interaction between virtual inheritance and dynamic typing increases class table overhead by 37.5%, while `npic` increases by 26.3%.

References

[1] Accredited Standards Committee. *Working Paper for Draft Proposed Internation Standard for Information Systems – Programming Language C++*, Nov. 1997. Doc. No. X3J16/97-0108 WG21/N1146.

[2] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 91–107, Austin, Texas, October 1995. ACM Press, New York, New York.

[3] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the Tenth European Conference on Object-Oriented Programming – ECOOP'96*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Linz, Austria, July 1996. Springer-Verlag.

[4] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '96*, pages 324–341, San Jose, California, October 1996.

[5] Michael Burke, Harini Srinivasan, and Peter F. Sweeney. A framework for evaluating space and time overhead for c++ object models. Technical Report RC 20421, IBM Thomas J. Watson Research Center, March 1996.

[6] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 397–408, Portland, OR, Jan. 1994.

[7] Microsoft Corp. and Digital Equipment Copr. *The Component Object Model Specification*, version 0.9 edition, Oct. 1995.

[8] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings of the Ninth European Conference on Object-Oriented Programming – ECOOP'95*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[9] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of the 1989 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 211–214, New Orleans, LA, October 1989. ACM Press, New York, New York.

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

[11] Mary F. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[12] Joseph (Yossi) Gil and Peter F. Sweeney. Space- and time-efficient memory layout for multiple inheritance. Submitted to OOPSLA'99, March 1999.

[13] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM TOPLAS, to appear*, 1999.

[14] Jr. Jonathan G. Rossie and Daniel P. Friedman. An algebraic semantics of subobjects. In *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '95*, page 187, Austin, Texas, Oct. 1995.

[15] D.T. Jones, M.J. O'Riordan, and M.J. Zbikowski. Method and system for implementing virtual functions and virtual base classes and setting a this pointer for an object-oriented programming language. US Patent 5,297,284, Mar. 1994. assignee Microsoft Corp.

[16] Michael Karasick. The architecture of montana: An open and extensible programming environment with an incremental c++ compiler. In *Proceedings of the Foundations of Software Engineering*, pages 131–142, Nov. 1998.

[17] Mark A. Linton and Douglas Z. Pan. Interface translation and implementation filterning. In *C++ Conference*, pages 227–236, Cambridge, MA, Apr. 1994.

[18] Stanley B. Lippman. *Inside the C++ Object Model*. Addison Wesley, New York, 1996. ISBN: 0-201-83454-5.

[19] Bruce Martin. The separation of interface and implementation in C++. In *C++ Conference*, pages 51–62, Cambridge, MA, Apr. 1994.

[20] Lee R. Nackman and John J. Barton. Base-Class Composition with Multiple Derivation and Virtual Bases. In *Sixth Usenix C++ Technical Conference*, 1994.

[21] Sara Porat, David Bernstein, Yaroslav Fedorov, and Joseph Rodrigue. Compiler optimizations of C++ virtual function calls. In *Proceedings of the Second Conference on Object-Oriented Technologies and Systems*, pages 3–14, Toronto, Canada, June 1996. Usenix Association.

[22] J. R. Rose. Fast dispatch mechanisms for stock hardware. In *Proceedings of the 1988 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 27–35, San Diego, CA, October 1988. ACM Press, New York, New York.

[23] Harini Srinivasan and Peter F. Sweeney. Evaluating virtual dispatch mechanisms for C++. Technical Report RC 20330, IBM Thomas J. Watson Research Center, Jan. 1996.

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

[25] P. F. Sweeney and F. Tip. A study of dead data members in C++ applications. In *In Proceedings of ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98), (Montreal, Canada)*, pages 324–333, June 1998. Also published as ACM SIGPLAN Notices 33(6).