Two-Dimensional Bi-Directional Object Layout

Joseph (Yossi) Gil[1]
Technion—Israel Institute of Technology
yogi@cs.technion.ac.il
and
William Pugh
Dept. of Computer Science
University of Maryland, College Park
pugh@cs.umd.edu
and
Grant E. Weddell
University of Waterloo
gweddell@math.uwaterloo.ca
and
Yoav Zibin
Technion—Israel Institute of Technology
yoav@zibin.net

Object layout schemes used in C++ and other languages rely on (sometimes numerous) compiler generated fields. We describe a language-independent object layout scheme, which is space optimal, i.e., objects are contiguous, and contain *no compiler generated fields* other than a single type identifier. As in C++ and other multiple inheritance languages such as CECIL and DYLAN, the new scheme sometimes requires extra levels of indirection to access some of the fields. Using a data set of 28 hierarchies, totaling almost 50,000 types, we show that this scheme improves field access efficiency over standard implementations, and competes favorably with (the non-space optimal) highly optimized C++ specific implementations. The benchmark includes an analytical model for computing the frequency of indirections in a sequence of field access operations. Our layout scheme relies on whole-program analysis, which requires about 10 micro-seconds per type on a contemporary architecture (Pentium III, 900Mhz, 256MB machine), even in very large hierarchies. We also present a layout scheme for separate compilation using the user-annotation of *virtual inheritance edge* that is used in C++ .

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types*; *structures*; G.4 [**Mathematical Software**]: Algorithm design; analysis

[1]Research supported in part by the generous funding of the Israel Science Foundation, grant No. 128/02.
Parts of the contribution of this paper were described by Pugh and Weddell [1993]; others were published in the proceedings of the 17^{th} Annual European Conference on Object-Oriented Programming (ECOOP'03).

General Terms: Algorithms, Design, Measurement, Performance, Theory
Additional Key Words and Phrases: Bi-Directional, Coloring, Hierarchy, Inheritance, Object Layout, Partially Ordered Sets

1. INTRODUCTION

A common argument raised by proponents of the single inheritance programming model is that multiple inheritance incurs space and time overheads and inefficiencies on the runtime system [Cargill et al. 1993; Magnussun et al. 1994]. A large body of research was targeted at reducing the multiple inheritance overhead in operations such as dynamic message dispatch and subtyping tests (see e.g., [Zibin and Gil 2001; 2002; 2003] for recent surveys). Another great concern in the design of runtime systems for multiple inheritance hierarchies is object layout that support efficient field access at runtime. To this end, both general purpose [Pugh and Weddell 1990] and C++ [Stroustrup 1997] language specific [Gil and Sweeney 1999; Eckel and Gil 2000] object layout schemes were previously proposed in the literature.

The various C++ layout schemes described in the literature are not space-optimal in the sense that they store in the layout *compiler generated fields*, i.e., fields whose sole purpose is to serve the runtime environment in accessing the principal object data. There is evidence [Eckel and Gil 2000] that such fields can be a significant space overhead.

Object space overhead can be traded for access time. The *field dispatching scheme* [Borning and Ingalls 1982], employed by many object oriented languages, recodes field access operations as calls to compiler generated accessor methods. In effect, the scheme reduces the problem of field dispatching into the more difficult problem of *method dispatching*. Although field dispatching does not increase the object size, it uses global tables which may be large, and incurs an access time penalty as a result of this reduction. Popular C++ layout schemes also have access time penalties, specifically accessing fields defined in virtual bases requires several memory dereferences. There are certain schemes which require $O(n)$ time for accessing a field defined in the n^{th} ancestor class.

This paper revisits the object layout problem in the general, language-independent setting. We propose a new object layout scheme that is space optimal, i.e., objects are contiguous, and contain *no compiler generated fields*. Hence, in terms of space, it is superior to previous C++ layout schemes. It is also superior to these in terms of field access efficiency to the (space-optimal) *field dispatching* scheme. In fact, in the new scheme the number of machine instructions required for accessing a field is bounded by a (small) constant. In the main implementation variant, most fields can be accessed in a single dereferencing instruction, while no field requires more than three such instructions.

In fairness, it should be said that these improvements are made possible by whole program analysis, whereas traditional C++ compilers are incremental. It is straightforward to adapt the new scheme to allow the dynamic addition of types with a single parent; the difficulty lies with the addition of types with multiple parents.

The scheme presented here is labeled as *two dimensional, bi-directional* since all objects can be thought of as being laid out first in a two-dimensional matrix, whose rows (also called *layers*) may span both positive and negative indices. The layout algorithm ensures that the populated portion of each such layer is consecutive, regardless of the object type. The particular object layout in one-dimensional memory is a cascade of these portions.

A data set of 28 hierarchies, totaling almost 50,000 types, was used in comparing the field access efficiency of the new scheme with that of different C++ specific layouts. Our analytical cost model shows that in this data set, the new scheme is superior to the standard C++ layout and to the simple inlining algorithm [Eckel and Gil 2000]. Even though the new layout is not C++ specific, it competes favorably in this respect with aggressive inlining [Eckel and Gil 2000], arguably the best C++ layout scheme.

1.1 Object Layout

To better understand the intricacies of object layout, consider Figure 1a, which depicts a small single inheritance hierarchy.

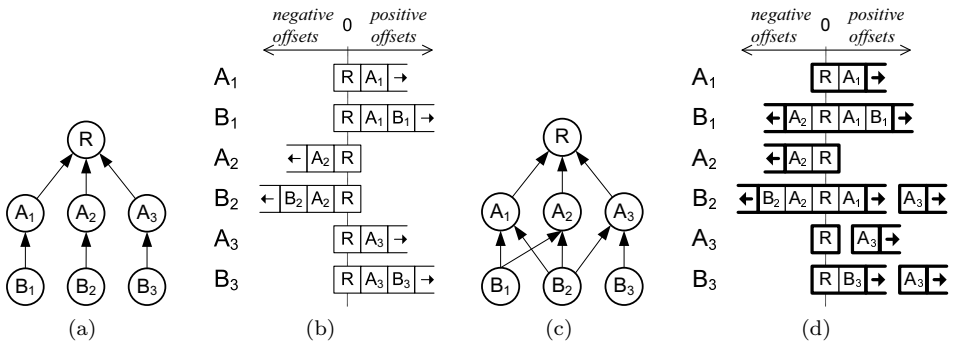


Fig. 1. A small single inheritance hierarchy (a), a possible object layout for this hierarchy (b), a multiple inheritance hierarchy (c), and a possible two-dimensional bi-directional layout for this hierarchy (d).

A possible object layout of the types defined in this hierarchy is shown in Figure 1b. This layout exploits the natural degree of freedom of memory, which allows populating it in either increasing or decreasing order. The fields of A_1 are laid out just after R. The layout of B_1 adds its own fields in increasing offsets. Types A_2 and B_2 are laid out in negative offsets. Types A_3 and B_3 have positive directionality.

Figure 1b demonstrates a degenerate case of the two-dimensional bi-directional layout scheme, in which there is only one layer. This layer is populated in both negative and positive offsets. In the general case, there are multiples layers, which may use for the same object type both positive and negative offsets, or even be empty.

Consider now the multiple inheritance hierarchy of Figure 1, obtained by adding multiple inheritance edges *(i)* from B_1 to A_2, and, *(ii)* from B_2 to A_1 and A_3.

Note: Here and henceforth, inheritance is assumed to be *shared* (`virtual` in the C++ jargon). Thus, in the figure, type B_2 has a *single* R sub-object. We believe that repeated inheritance, i.e., where type B_2 has two R sub-objects, is a rarity, or as one once wrote: "repeated inheritance is an abomination".[2] Repeated inheritance also causes a tight coupling between message dispatching and object layout (e.g., in the presence of two R sub-objects, which field should a method access?), and is therefore not supported in our layout scheme.

1.2 The Strong Conformance Requirement

It is easy to see that the layout of Figure 1b can be extended so that despite having two parents, type B_1 still has a contiguous layout. This layout is obtained by placing its A_2 sub-object just prior to the R sub-object.

On the other hand, the layout of B_2 becomes difficult, since at the same positive offsets immediately following R we expect to find both the fields of A_1 and the fields of A_3. This difficulty is no coincidence, and is in fact a result of two conflicting requirements which we implicitly made:

> **(i) The strong conformance requirement (or fixed offsets [Pugh and Weddell 1990]):** Every type must be laid out at the same offset in all of its descendants, and
> **(ii) Contiguity requirement:** A layout cannot contain "holes", i.e., for every type, the offsets of its ancestors define an interval.

If requirement (i) holds then field access is as efficient as record access: a single load instruction. If requirement (ii) holds then objects do not have any additional dynamic memory overhead due to object layout. (Due to subtyping tests and method dispatching, most languages add a type identifier to the fields of the hierarchy's root.) Phrased differently, if requirement (ii) does not hold for a type A then at runtime every instance of A will have a hole, and the amount of wasted storage depends on the number of instances of A.

If the layout of A_1, A_2 and A_3 in Figure 1c is required to be contiguous, then the fields of each of these types must be laid out adjacent to R. Since the layout of R in memory has only two sides, then it must be that at least two of A_1, A_2 and A_3 are laid out at the same side of R. This is not a problem as long as these two types are never laid out together, as is the case in single inheritance. The difficulty is raised in multiple inheritance, specifically, when there is a common descendant of these two types. More precisely, we can say that two types are in *conflict* if neither is a descendant of the other, yet, they have a common descendant.

No types are in conflict in a single inheritance hierarchy. In Figure 1c there are three conflicts: between A_1 and A_2 (due to both B_1 and B_2), A_1 and A_3 (due to B_2), and between A_2 and A_3 (due again to B_2).

The following simple argument shows that in the case of a single layer and a common ancestor to the hierarchy, two conflicting types cannot have the same directionality without violating either the strong conformance requirement, or having a type with non-contiguous layout: Assume by contradiction that the two conflicting types have the same directionality. They are obviously laid out at *different*

[2] words of an anonymous reviewer to [Gil and Sweeney 1999]

(say) positive offsets in their common descendant. Consider now the layout of the type which is allocated the greater offset. This layout will have a hole at the offset of the other type.

We therefore strive to assign opposing directionality to conflicting types. A contiguous and strongly conforming layout exists for B_1 since its conflicting parents have opposing directionality. However, there is no way of assigning opposing directionality in all three conflicts of our example.

Figure 1d shows a possible two-dimensional bi-directional layout for the hierarchy of Figure 1c. This layout maintains contiguity by sacrificing the strong conformance requirement. This is achieved by laying out objects in one or more layers, where each layer uses a bi-directional layout. Figure 1d uses two layers (marked in bold), e.g., type B_3 has three ancestors $\{\mathsf{R}, \mathsf{A}_3, \mathsf{B}_3\}$, where $\{\mathsf{R}, \mathsf{B}_3\}$ are in the first layer and $\{\mathsf{A}_3\}$ in the second. Note that for some types the second layer is empty, e.g., in type B_1. Recall that there are three conflicting types $\{\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3\}$. Our layout places two conflicting types in either different layers (e.g., A_1 and A_3) or in different directionality (e.g., A_1 and A_2). Within each layer we still maintain strong conformance, e.g., the offset of A_1 within the first layer is always 1, and the offset of A_3 within the second layer is always 0. Observe that the offset of the second layer is different in different types. Section 8 explains how to find at runtime the offset of a layer within a type, and Section 5 presents the algorithm that calculates the layout.

Note that separate compilation discovers too late that two types are in conflict, i.e., after the layout of these two types was determined. For this reason, our layout scheme, just as all other optimizing layouts (including aggressive inlining), relies on whole program analysis.

1.3 Evaluation Criteria

A layout scheme is evaluated by the following criteria.

(1) *Dynamic memory overhead.* This is extra memory allocated for objects, i.e., memory beyond what is required for representing the object's own fields. Ideally, this overhead is zero. However, holes in a noncontiguous object layout contribute to this overhead. Another overhead of this kind are compiler generated fields, e.g., virtual function table pointers (VPTRs) in C++ .
Note that the semantics of most object oriented languages dictates that the layout of each object includes at least one *type identifier*. This identifier is used at runtime to identify the object type, for purposes such as dynamic message dispatch and subtyping tests. This identifier can be conveniently thought of as a field defined in the common root type , and therefore is not counted as part of the dynamic memory overhead. However, if a scheme allocates multiple type identifiers, as is the case with the C++ standard layout, then all but the first identifier contribute to this overhead.

(2) *Field access efficiency.* This is the time required to realize the field access operation `o.f`. Ideally, fields can be accessed in a single machine instruction, which relies on a fixed offset (from the object base) addressing mode. Layout schemes often rely on several levels of indirection for computing a field location in memory.

Common practice is to lay out consecutively all fields introduced in a certain type. Since `f` is supplied at compile time, the type t' in which `f` was introduced can be pre-computed. The main duty of the runtime system is to find the location in memory in which the fields of t' are laid out in t, the type of `o`.

(3) *Static memory overhead.* These are the tables and other data-structures used by the layout which are shared between all objects of a certain type. This overhead is usually less significant than the dynamic memory overhead, and therefore it seems worthwhile to maximize sharing. On the other hand, retrieving the shared information comes at the cost of extra indirections, and may reduce field access efficiency.

(4) *Time for computing the layout.* This is the time required for computing the layout, which could be exponential in some schemes.

Outline Section 2 defines the object layout problem, and then describes the criteria used in evaluating object layout schemes. These criteria are then used in Section 3 to compare our result in the context of previous work.

A formal definition of a uni-directional two dimensional layout is the subject of Section 4. Section 5 presents the algorithm for computing the actual layout, which is adapted for separate compilation in Section 6 The correctness and completeness of these algorithms are the subject of Section 7, which also characterizes the complexity of the problem. Section 8 suggests three different strategies for realizing the formal layout as a bi-directional layout scheme.

Section 9 describes the data-set used in the benchmark, while Section 10 gives the experimental results. Finally, conclusions and directions for future research are given in Section 11.

Appendix A summarizes the terminology and notation used in this paper.

2. THE OBJECT LAYOUT PROBLEM

2.1 Definitions

Leading to a more exact specification of the problem, we must first make precise notions such as a hierarchy, incomparable types, and introduced and accessible fields in a type.

Formally, a *hierarchy* is specified by a set of types $\mathcal{T}$, $n = |\mathcal{T}|$, and a partial order, $\preceq$, called the *subtype relation* which must be reflexive, transitive and anti-symmetric. Let $a, b \in \mathcal{T}$ be arbitrary types. Then, if $a \preceq b$ holds we say that a is a *subtype* of b, that b is a *supertype* of a, and that a and b are *comparable*. (If neither $a \preceq b$ nor $a \succeq b$ holds, we say that the types are *incomparable*.) Also, if there does not exist c such that $a \preceq c \preceq b$ and $c \neq a$, $c \neq b$, then we say that a is a *child* of b and that b is a *parent* of a. A hierarchy is *single inheritance* if each $a \in \mathcal{T}$ has at most one parent, and *multiple inheritance* otherwise.

Types a and b are *conflicting* if they are incomparable and there exists a type $c \in \mathcal{T}$, $c \preceq a$ and $c \preceq b$. If two conflicting types are given the same memory address, then they will collide in the layout of c.

The set of ancestors of a type $a \in \mathcal{T}$ is $\text{ancestors}(a) \equiv \{b \in \mathcal{T} \mid a \preceq b\}$. We denote the number of ancestors of a by θ_a. Note that $a \in \text{ancestors}(a)$.

An instance of the *object layout problem* is specified by a hierarchy of types, each *introducing* fields, which can be thought of as unique names or selectors. For a type $t \in \mathcal{T}$, let $|t|$ denote the memory size required for fields introduced in t.

The *accessible* fields of a type include all fields introduced in it and in any of its proper supertypes. Our objective is to design a *layout scheme* for the objects of each of the types in the hierarchy, and a method for locating each of the accessible fields of this type at runtime. Specifically, given an object address `o` of type t, and `f`, an accessible field of t, the runtime system should be able to compute the address of `o.f`. The selector `f` is a compile time constant, while `o` is supplied only at runtime.

2.2 Assumptions

We shall assume that all hierarchies are rooted, i.e., that there exists a type $r \in \mathcal{T}$, such that for all $a \in \mathcal{T}$, $a \preceq r$. For example, type `R` is the root of the hierarchy depicted in Figure 1. We also assume that each object has within a fixed offset a unique type-identifier (type-id), which is also needed for the purposes of method dispatching and subtyping tests. (The type-id can be any kind of unique pointer such as C++ 's virtual table pointer.) It is convenient to think of the type-id as a field introduced in the root r. (Although C++ does not have a common root, we can artificially add a common root that introduces the type-id.)

Observe that the process of method dispatching is similar to field access: Given a compile time constant (the method name) and an object at runtime, the method dispatcher must find the correct implementation of the method. Yet, as we commented earlier, field access is easier than method dispatching since (by wide agreement) field overriding is disallowed in most languages. The reason for this is that unlike methods, fields have no body which can be re-implemented in an overriding definition. One may contemplate overriding the *type* definition of a field, but this cannot be done without introducing intriguing type safety issues: on one hand, since fields can be read, a safe redefinition must be co-variant; on the other, since fields can be written, a safe redefinition must be contra-variant.

We shall therefore assume that a field name can only be used once in each type. Stated differently, our demand is that no run time dispatching process is required to select the particular "implementation" of a field name. This is precisely the case in statically typed languages, where the field name and the static object type uniquely determine the introducing class. (Note that although there are languages (e.g., C++ and JAVA [Arnold and Gosling 1996]) which allow field *overloading* in that a derived class may reuse the name of a **`private`** field defined in a base class, our assumption is trivially satisfied by a simple renaming.)

Some languages such as SMALLTALK [Goldberg 1984] restrict field access to the methods of the defining object, i.e., all fields are **`private`** and never **`protected`** nor **`public`**. In such languages the strong conformance requirement does not need to be satisfied, and the object layout problem becomes trivial, even with the face of multiple inheritance.

Observe also that in a dynamically typed language that supports non-private fields, there must be a runtime check that the accessed field is defined in the object. Such checks are related to subtyping tests and even to a more general dispatching problem which have received extensive coverage in the literature [Zibin and Gil

2001; 2002; 2003], and shall therefore be excluded from the domain of discourse.

3. PREVIOUS WORK

Object layout in a single inheritance hierarchy can simultaneously optimize all the above metrics. As can be seen in Figure 1b, both static and dynamic memory overheads are zero. Field access efficiency is optimal with no dereferencing. Also, the computation of the layout is as straightforward as it can be.

A trivial layout scheme for multiple inheritance which maintains the strong conformance requirement is that the layout of each type reserves memory for *all* fields defined in the hierarchy. Static memory overhead, time for computing the layout, and field access efficiency are optimized. However, dynamic memory overhead is huge since each object uses memory of size $\sum_{t \in \mathcal{T}} |t|$, regardless of its actual type, which usually has far fewer accessible fields.

Pugh and Weddell [1990] investigated more efficient layout schemes which still fulfill the strong conformance requirement, and therefore must sacrifice the contiguity requirement, i.e., objects might have "holes" in their layout. The dynamic memory overhead of their main bi-directional object layout scheme is only 6% in one case study, compared to 47% in a unidirectional object layout. The authors also showed that the problem of determining whether a contiguous bi-directional layout exists is NP-complete.

At the other extreme stands what may be called the *field dispatching* layout scheme, which is employed by many dynamically typed programming languages including CECIL [Chambers 1993] and DYLAN [Shalit 1997]. In this scheme, the layout of type t is obtained by iterating (in some arbitrary order) over the set ancestors(t), laying out their fields in order. Since the strong conformance property is broken, the scheme must encapsulate fields in accessor methods. If a field position changes in a subtype, its accessor method is overridden. The dynamic memory overhead in this scheme is zero.

Dispatching on accessor methods can be implemented by an $n \times n$ *field dispatch matrix* which gives the base offset of a type in the layout of any of its descendants. This static memory overhead can be reduced if the matrix is compressed by e.g., techniques used for method dispatching. The SMARTEIFFEL compiler [Zendra et al. 1997] avoids using a dispatching matrix by using an inlined binary search over the type-id. See, e.g., [Zibin and Gil 2002], for a recent survey of dispatching techniques.

The main drawback of field dispatching is in reduced field access efficiency. In the matrix implementation, field access requires at least three indirections in the simplest version, and potentially more with a compressed representation of the matrix.

Myers [Myers 1995] suggested a bi-directional layout for THETA which have a similar restriction as in JAVA regarding classes and interfaces: a class can extend only a single superclass but it can implement several types. With this restriction the layout problem becomes trivial (but method dispatching is still a challenge). Myers also presents two variants of the technique for multiple inheritance: (i) either recompile the code of the superclasses as described below in SMARTEIFFEL, or (ii) use field dispatching (called *dynamic field offset*) with an uncompressed field dispatching matrix.

3.1 SmartEiffel's layout

SmartEiffel compiler (which used to be called SmallEiffel [Zendra et al. 1997]) *specialize* the code of the superclass, i.e., all methods are overridden and the code of the superclass is copied (by the compiler) to the subclass. The layout of a type does not necessarily conform to the layout of its ancestors. Consider a method in type t that accesses a field f defined in a type t'. When $t \preceq t'$ then the code of that method is unique for t and the offset of f is thus known statically, making field access as efficient as a record access. However, when $t \not\preceq t'$ then the compiler inserts a dispatching method that finds the offset of the field according the type-id of t. Since most field accesses are of the former type, field access has good efficiency.

Specialization enables many optimizations due to the fact that the exact type of `this` is known statically, such as efficient field access and avoiding dispatching for method calls on `this`. However, there is an exponential blow-up of the code-space (since the code of a class is copied to all its descendants), which can affect runtime performance due to code caching. Also, the efficiency of method dispatching is degraded since a method is now overridden in *all* descendants. For example, consider a method m that is defined only in a single type t and never overridden in any descendant. Then an optimizing compiler can avoid dispatching on that method, and use a direct jump. However, in SmartEiffel, method m is overridden in *all* descendants due to specialization, forcing the compiler to insert code that binary search over the type-id of all descendants. (Note that in order to have efficient field access in m we *must* override this method in all descendants, since fields' offset can be different in different descendants.)

3.2 C++ 's layout

An interesting tradeoff between the two extremes is offered by the memory model of C++ [Lippman 1996]. C++ distinguishes between `virtual` and *non*-`virtual` bases. We are not so interested in the textbook [Stroustrup 1997] difference between the two. Instead, we say that a type is a *virtual base* if two or more of its children have a common descendant. For non-virtual bases, C++ uses a relaxed conformance requirement. Let $t_1, t_2, t_3 \in \mathcal{T}$ be such that t_1 is a non-`virtual` base of t_2, and t_3 is an arbitrary subtype of t_2.

> **The weak conformance requirement:** The offset of t_1 with respect to t_2 is fixed in all occurrences of t_2 within $t_3 \preceq t_2$.

In other words, although the offset of t_1 is not the same in all of its descendants, it is fixed with respect to any specific descendant t_2, regardless of where that descendant is found. Consequently, to find where t_1 is located within t_3 it is sufficient to find the address of t_2 within t_3.

The weak conformance requirement can be maintained together with object contiguity in many multiple inheritance hierarchies, specifically those with no virtual-bases. In such hierarchies, e.g., the hierarchy in Figure 2a,

> *The layout of a type is the concatenation of the layout of its parents followed by its fields.* (1)

The cells with a dot in Figure 2b represent a type-identifier (VPTR in the C++ jargon). (For simplicity of the presentation, we assume that all classes in the

hierarchy have at least one *virtual function*, and thus each object must have a VPTR in order to perform message dispatching.)

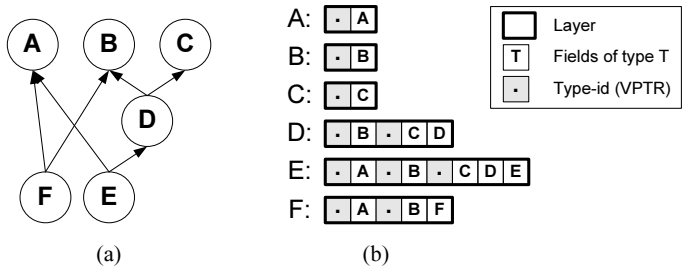


Fig. 2. A type hierarchy without virtual bases (a) and its C++ layout (b).

Consider the C++ layout in Figure 2b. According to (1), the layout of E is the concatenation of the layout of A, than the layout of D, and than the fields of E itself. Note how this layout satisfies the weak conformance requirement for $t_1 = \mathsf{B}$ and $t_2 = \mathsf{D}$: the offset of B is fixed in all occurrences of D within $t_3 \preceq \mathsf{D}$, e.g., when $t_3 = \mathsf{D}$ we see that B has the first offset and when $t_3 = \mathsf{E}$ then still B has the first offset in the layout of D within E. The offset of B within D will be the same in all descendants of D due to (1) which uses the layout of D without modifying it. Note that the offset of B is different in F (which is not a descendants of D). Thus *strong conformance* is not maintained, i.e., a type is not always located at the same offset, making it necessary to apply a process called **`this`**-adjustment [Stroustrup 1994] in order to access a field introduced in a supertype. For example, a method of B cannot be invoked on an object of type E, without first correcting the pointer to the object (**`this`** in the C++ jargon), coercing it to type B.

Observe in Figure 2b that each layout starts with a VPTR, and some layouts have more than one VPTR. The reason for having several VPTRs is the requirement that a VPTR must be located within a fixed offset from **`this`** in order to perform subtyping tests and message dispatching (in C++ , this offset is zero, i.e., **`this`** always points to a VPTR). Therefore, after correcting the pointer from E to B, that pointer must still point to a VPTR, forcing C++ to place a VPTR before the fields of B. Note that these VPTRs cannot be shared among several instances of the same class and thus contribute to *dynamic* memory overhead.

The **`this`**-adjustment model incurs many penalties other than dynamic memory overhead and the time required for the addition in correcting **`this`**. For example, the runtime system must apply null checks before **`this`** can be corrected. Also, a conversion from an array of subtypes to an array of supertypes cannot be done in constant time. Finally, the pointers to the same object may have different values which is a serious hurdle for *garbage collectors* and for efficient *identity testing*.

Object layout becomes much more complex in the presence of virtual bases, as demonstrated by the hierarchy in Figure 3a. Type A is a virtual base since it has two children (B and C) with a common descendant D. C++ allows a programmer to denote some of the inheritance edges as **`virtual`** (see note below on separate compilation). In the figure, the dashed edges $\langle \mathsf{B}, \mathsf{A} \rangle$ and $\langle \mathsf{C}, \mathsf{A} \rangle$ are virtual so that D

has a single A sub-object. Recall that we assumed that all inheritance is *shared* (never *repeated*), thus the programmer must annotate all edges $\langle b, a\rangle$ as virtual whenever there exists a child c of a that share a common descendant d with b, i.e.,

$$d \preceq c \preceq a \text{ and } d \preceq b \preceq a \text{ but } b \not\preceq c \text{ nor } c \not\preceq b. \tag{2}$$

Such a quartet $\langle a, b, c, d\rangle$ is called a *diamond* due to diamond shape in the inheritance hierarchy. Observe that Figure 2a has no diamonds, and thus no edge needs to be marked as virtual.

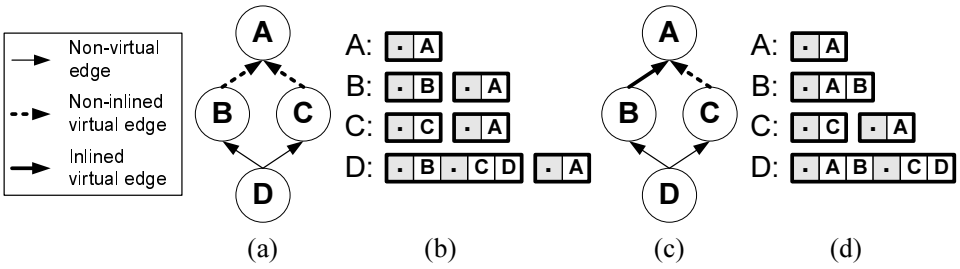


Fig. 3. A diamond hierarchy (a), its C++ layout (b), the hierarchy with the inlined virtual edge $\langle$B, A$\rangle$ (c), and its inline C++ layout (d).

If we "ignore" the virtual edges, then the resulting hierarchy has no diamonds and can be laid out using rule (1). Figure 2b shows the C++ layout of Figure 2a. Note that all classes (except A) have two *layers*. C++ uses *virtual base pointers* (VBPTRs) to tie the layers of the same object, e.g., in order for a method of B to access a field of A it must traverse a VBPTR. Gil and Sweeney [1999] give a detailed description of VBPTRs. We only mention that VBPTR can be stored directly in the objects, as in the "standard" C++ implementation, contributing to dynamic memory overhead, or moved to the static memory, at the cost of increasing field access time. (For that reason, VBPTRs were not drawn in Figure 3.) Also, in order to be able to access fields at constant time, an implementation must store (a potentially quadratic number of) *inessential* VBPTRs. We note that referencing fields through VBPTRs also requires **`this`**-adjustment, and that a virtual base must also have a VPTR.

An optimizing compiler (with whole program information) can decide to *inline* a subset of virtual edges, e.g., the bold edge $\langle$B, A$\rangle$ was inlined in Figure 2c. Note that even after inlining this edge we can still use rule (1) for doing the layout. After inlining, the remaining set of virtual edges E' must satisfy

$$\begin{array}{l}\text{There are no two edges } \langle a, b\rangle, \langle a, c\rangle \in E \setminus E' \\ \quad \text{where } b \text{ and } c \text{ has a common descendant } d.\end{array} \tag{3}$$

Phrased differently, for every diamond $\langle a, b, c, d\rangle$ either $\langle a, b\rangle \in E'$ or $\langle a, c\rangle \in E'$, i.e., we cannot inline two virtual edges that participate in a diamond. Observe that in the inline C++ layout of Figure 2d is more efficient than that of Figure 2b since only the layout of C has two layers.

Gil and Sweeney [1999] proposed several optimizations of the standard C++ layout, which were then empirically evaluated by Eckel and Gil [2000], whose main yardstick was dynamic and static memory overhead. The main optimization which contributes to field access efficiency is *simple-inline* which tries to reduce the number of virtual bases by conforming transformations of the hierarchy. *Aggressive-inline* does the same, using a maximal independent-set heuristic as procedure for finding a close to optimal set of transformations. The *bi-directional object layout* optimization reduces dynamic memory overhead but does not contribute to field access efficiency.

For the purpose of illustration, Figure 4 depicts a type hierarchy and its aggressive-inline C++ layout. (This hierarchy will be used as our running example: in Section 8.4 for demonstrating two-dimensional bi-directional, and then in Section 5 for gaining intuition into the algorithm that generates this layout.)

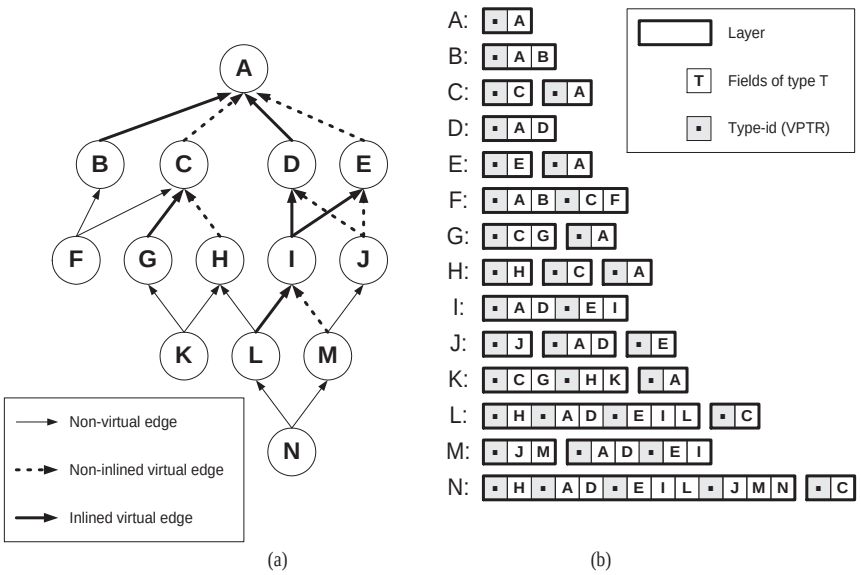


Fig. 4. A type hierarchy (a) with its aggressive-inline C++ layout (b).

Separate compilation: It is well known that the C++ compiler is incremental in the sense that a layout of a class depends only on the layout of its superclasses, i.e., the compiler does not require knowledge of any future *descendants*. However in the presence of virtual bases, layout must be guided by user-defined "predictions", and will therefore be called *prediction-guided layout*. Recall that the inheritance edges $\langle \mathsf{B}, \mathsf{A} \rangle$ and $\langle \mathsf{C}, \mathsf{A} \rangle$ in Figure 3a are virtual so that D has a single A sub-object. Marking the edge $\langle \mathsf{B}, \mathsf{A} \rangle$ as virtual is in fact a *prediction* given to the compiler so it will know of a future conflict in a *descendant* of B. In other words, the compiler does not need to know the set of *descendants* of a type since it is given predictions on future conflicts instead. Section 6 presents a variant of our algorithm for prediction-

guided layout, i.e., a subset of the edges that satisfy (3) is marked as *virtual* and the compiler must calculate the layout of a type based only on its set of ancestors.

Comparison with two-dimensional bi-directional: Our two-dimensional bi-directional scheme incurs *no dynamic memory overhead*. In this respect it is at least as good as any other layout scheme, and strictly better than all C++ implementations (which may include more than one VPTR). (See [Sweeney and Burke 2003] for a study of the space overhead in real C++ programs.) The most interesting criterion for comparison with C++ and field dispatching is therefore field access efficiency. We shall see that our new scheme competes favorably even with the highly optimized and language specific aggressive-inline layout scheme.

Our results indicate that the time for computing the new layout is small—about 10 μSec per type (see Section 10). We also find that the static memory overhead is small compared both to field dispatching and various C++ techniques.

The new layout is *uniform*, in the sense that (unlike C++) the runtime system does not need any information on the static type of an object pointer in order to access any of its fields. Consider an object `o` and a field `f`. Then, the sequence of machine instructions for the field access operation `o.f` depends only on the selector `f`, and is the same regardless of the type of `o`. This is in contrast to languages such as C++ in which, depending on the static type of `o`, access to field `f` is either direct, or through indirection.

4. A TWO-DIMENSIONAL LAYOUT

We present our two-dimensional bi-directional scheme in five stages. In this section, we explain the notion of a formal *uni-directional* two dimensional layout, or for short, *two-dimensional layout*. Section 5 presents the algorithm for computing the two-dimensional layout, which is adapted for separate compilation in Section 6. The correctness and completeness of these algorithms are the subject of Section 7. Finally, Section 8 shows how the two-dimensional layout is made bi-directional and gives the actual representation in memory.

DEFINITION 4.1. *A* two dimensional layout *of $\mathcal{T}$ is a pair of two functions $\langle \phi, \rho \rangle$, such that $\phi, \rho : \mathcal{T} \mapsto \{1, 2, \ldots\}$.*

A two dimensional layout allocates to each type $t \in \mathcal{T}$ a two-dimensional *address* $\langle \phi(t), \rho(t) \rangle$:

—Coordinate $\phi(t)$, $1 \leq \phi(t) \leq \Phi$, is the *semi-layer* of type t, where Φ is the number of semi-layers used by the layout.

—Coordinate $\rho(t) \geq 1$ is the integral *position* of t in its semi-layer.

The uni-directional *semi-layers* will be joined together in pairs in Section 8 to make bi-directional *layers*.

The fields introduced by a type t are positioned in the address defined by $\langle \phi(t), \rho(t) \rangle$. This positioning persists in all descendants of t. Dually, we can make the following definition.

DEFINITION 4.2. *The* entire layout of a type t, $L(t)$ *is specified by the multi-set of addresses of t and its ancestors, i.e.,*

$$L(t) = \{\langle \phi(t'), \rho(t') \rangle \mid t' \in \text{ancestors}(t)\}.$$

There is a very simple two-dimensional layout which gives distinct addresses to all types. This *field-dispatching layout* is realized by letting $\Phi = |\mathcal{T}|$ and positioning the i^{th} type at the first position in the i^{th} semi-layer.

Reducing the number of semi-layers Φ is desirable for several reasons that will be further explained in Section 8: (i) the static memory overhead is $\Phi/2 - 1$ bytes per type, and (ii) field access becomes more efficient as the number of semi-layers decreases. To achieve this objective, a layout should reuse the same address for different types. There are however limitations to such a reuse.

DEFINITION 4.3. *The layout of a type t is* collision-free *if all addresses in $L(t)$ are distinct, i.e., $L(t)$ is a set rather than a multi-set. A layout of a hierarchy is collision-free if the layout of all types in it is collision-free.*

In requiring that the layout of a certain type is collision-free, we require first that the fields introduced in the type do not collide with the fields of any of its ancestors. Moreover, we ensure that there are no collisions between the (proper) ancestors themselves of a type, as may be the case in multiple inheritance. Stated differently, this requirement re-iterates what we have observed before: that two types which are in conflict must be allocated to distinct addresses.

Henceforth, we consider only collision-free layouts. We shall also require that no "holes" occur in the layout of any particular type; that is, that no type is located at a position of a semi-layer unless all previous positions of this semi-layer are populated by an inherited type.

DEFINITION 4.4. *The layout of a type $t \in \mathcal{T}$ is* contiguous *if the set $L(t)$ satisfies the property that if $\langle \sigma, p \rangle \in L(t)$ and $p > 1$, then $\langle \sigma, p-1 \rangle \in L(t)$. A layout of a hierarchy is contiguous if the layout of all types in it is contiguous.*

A collision-free and contiguous two-dimensional layout always exists, e.g., the field-dispatching layout. Also, in the single-inheritance setting, there is a collision-free and contiguous layout scheme which uses a single semi-layer: A type at height i in the inheritance *tree* is laid out at the i^{th} position. Note that in single-inheritance types are laid out in descending subtyping order. A similar property will hold for multiple inheritance as well.

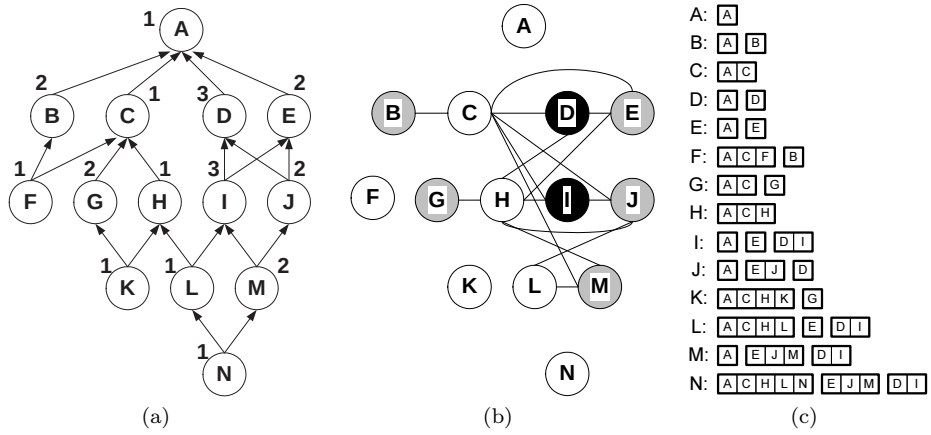
In the next three sections we turn to the problem of finding such a layout in the multiple-inheritance setting, which minimizes Φ, the number of layers. We will then discuss the implementation of a collision-free and contiguous layout.

5. COMPUTING TWO-DIMENSIONAL ADDRESSES OF TYPES

This section is dedicated to the issue of finding a two-dimensional layout, i.e., finding two functions ϕ and ρ which define a layout which is both collision-free and contiguous. As explained above, two such functions always exist. The challenge is to find two such functions which achieve the smallest possible number of semi-layers.

We will see below in Section 7 that the problem of finding the minimal number of semi-layers is NP-complete. Therefore, this section will be settled with a heuristic, which as will be shown in Section 10, performs well in practice.

Returning to our running example, Figure 5a shows the allocation of types in the hierarchy of Figure 4 to semi-layers.



(a) (b) (c)

Fig. 5. (a) The allocation of types in the hierarchy of Figure 4 to semi-layers, (b) the conflict graph with its coloring, and (c) the resulting two dimensional layout.

The figure shows the partition of types into the three semi-layers as used by the layout: seven types A, C, F, H, K, L, and N are in semi-layer 1. Semi-layer 2 includes five types: B, E, G, J, and M. Types D and I are in semi-layer 3.

In making such a partitioning, the algorithm must be concerned with the question of whether two given types $a, b \in \mathcal{T}$ can be allocated to the same semi-layer, and what their relative ordering in that semi-layer should be.

—Suppose first that the two inputs are comparable. Then, without loss of generality, $a \preceq b$. We have that whenever a appears in the layout of any arbitrary type, so does b. Therefore, with the absence of other constraints, we can allocate a and b into the same semi-layer, and a must be placed after b in this semi-layer.

—If however a and b are incomparable, then they could be allocated to the same semi-layer, and even to the same position in the level, as long as they do not occur together in the layout of any third type c. In other words, the allocation is allowed as long as a and b have no common descendants, i.e., a and b are not in *conflict*.

Figure 5b shows the *conflict graph* of our running example. In this graph, two types are connected by an edge if they are in conflict.

We see in the figure that no edges are incident on A. This is because A is the root type, and as such is comparable with all types in the hierarchy. Also, no edges are incident on the leaves F, K and N. On the other hand, the edge between C and E (for example) is due to their common descendant L.

Algorithm 1 shows how the conflict graph of a given hierarchy is computed.

Lines 1–10 compute the edges in the conflict graph. In the main loop, we consider the ancestors of each candidate t. There is a conflict between any two of its ancestors p_1 and p_2 if they are incomparable. For example, when $t = \mathsf{L}$, $p_1 = \mathsf{C}$, and $p_2 = \mathsf{E}$ we discover that neither $p_1 \not\preceq p_2$ nor $p_1 \not\succeq p_2$, and we therefore add the edge $\{\mathsf{C}, \mathsf{E}\}$ to the conflict graph.

A node coloring of the conflict graph provides a collision-free allocation. We of course seek a minimal coloring of this graph.

Algorithm 1 Compute the conflict graph of a hierarchy

Given a hierarchy $\mathcal{T}$ compute its conflict graph G.

```
1: Let E ← ∅ // E is the set of edges in the undirected conflict graph
2: For all t ∈ T do // Consider all possible common descendants
3:    For all p1, p2 ∈ ancestors(t) do // p1 and p2 have a common descendant t
4:       If p1 ⋠ p2 and p1 ⋡ p2 then // p1 and p2 are incomparable
5:          If {p1, p2} ∉ E then // A new conflict edge found
6:             E ← E ∪ {{p1, p2}}
7:          fi
8:       fi
9:    od
10: od
11: Let G ← ⟨T, E⟩
```

Figure 5b also gives a coloring of the conflict graph of the running example. A total of three colors are used: white, grey and black. A side-by-side comparison of Figure 5b with Figure 5a reveals that white nodes are allocated to semi-layer 1, grey to semi-layer 2, and black to semi-layer 3.

Algorithm 2 shows how the two-dimensional layout of a given hierarchy is found. The algorithm computes the number of semi-layers, and, for each type in the input, its semi-layer and its position within this semi-layer.

Algorithm 2 Produce the two-dimensional layout of a hierarchy from its conflict graph

Given a hierarchy $\mathcal{T}$ and its conflict graph G, return Φ, the number of semi-layers, and $\phi(t)$ and $\rho(t)$ for each type $t \in \mathcal{T}$

```
1: Let φ : T ↦ [1, ..., Φ] be a coloring of the nodes of G

2: For all t ∈ T do // Compute the position of t in its semi-layer
3:    ρ(t) ← 0 // Start counting ancestors in the same semi-layer as t
4:    For all t' ∈ ancestors(t) do
5:       If φ(t') = φ(t) then // Ancestor t' is in the same semi-layer as t
6:          ρ(t) ← ρ(t) + 1
7:       fi
8:    od
9: od
```

Running Algorithm 2 on the hierarchy of Figure 4 with the coloring of Figure 5b, produces the two dimensional layout in Figure 5c. Note that the layout in Figure 5c is both contiguous and collision-free, which will be proven in Theorem 7.1. Observe that the number of semi-layers is the same as the number of colors, which is 3. Also observe how types are allocated positions in descending subtyping order, e.g., in the layout of N the positions in the first semi-layer are allocated in descending subtyping order $\mathsf{A} \succeq \mathsf{C} \succeq \mathsf{H} \succeq \mathsf{L} \succeq \mathsf{N}$.

There are two main steps to Algorithm 2: First, line 1 finds an assignment of types to semi-layers, while the main loop in lines 2–9 computes the position of each type in its semi-layer:

(1) The first step employs a simple, greedy heuristic to color the graph. This heuristics iterates over the nodes in order of descending degree, and assigns to each the first color which was not previously assigned to any of its neighbors. A favorable property of this heuristic is that the sizes of color groups decrease quickly, i.e., $\phi^{-1}(1)$ is the largest set, $\phi^{-1}(2)$ much smaller than it, etc. This is important for our application since fields in the first layer can be accessed in a single indirection (see Section 8), therefore this layer should be as large as possible.

(2) To ensure the contiguous property of the layout, each type is positioned immediately after the subset of its ancestors which are assigned to the same semi-layer. Thus, lines 3–8, count in $\rho(t)$ the number of ancestors t', such that t' is assigned in the same semi-layer as t. Positions $1, \ldots, \rho(t) - 1$ in this semi-layer will be occupied with all such proper ancestors, while t is positioned in $\rho(t)$.

6. PREDICTION-GUIDED LAYOUT

The subject of this section is a variant of Algorithm 1 and Algorithm 2 for *separate compilation*, i.e., the layout of a type t can be determined solely from information on its ancestors $t' \succeq t$. The challenge in separate compilation is maintaining any kind of conformity requirement in the presence of unknown future descendants. *Field dispatching* places the fields of each supertype in a different layer (see Section 3), thus the number of layers in a type is exactly the number of its ancestors. This is the only technique which fits the model of separate compilation since the layers can have different offsets in different descendants, i.e., there is no conformity requirement which can be broken by future descendants.

In contrast, C++ has the weak conformity requirement which can always be broken by future descendants, unless *all* inheritance edges are marked as virtual. To see why all edges must be marked as virtual, assume by contradiction that there was an edge $\langle a, b\rangle$ which was not marked as virtual. We now add two future descendants: c (that extends a) and d (that extends b and c), which creates a *diamond* $\langle a, b, c, d\rangle$ (see Section 3.2). The semantics of C++ dictates that in such a case we have repeated inheritance semantics meaning that d have two a sub-objects. However, recall that we assume that all inheritance is shared, i.e., the semantics of repeated inheritance is never desired. Type d cannot change the semantics to shared without changing its supertype b by declaring the edge $\langle a, b\rangle$ as virtual.

When all edges are marked as virtual in C++ then the number of layers in a layout of a type is exactly the number of its ancestors, which reduces C++ layout to field dispatching. (In fact C++ 's efficiency is worst due to `this` adjustment.) Thus a C++ programmer joggles two conflicting forces: (i) flexibility that dictates that all edges are marked as virtual, and (ii) efficiency that dictates that no edge is marked as virtual. Consider an inheritance edge $\langle a, b\rangle$. The programmer attempts to *predict* if future descendants of b might inherit a through a different path, i.e., will there be a *diamond* that includes this edge. If so, then we declare $\langle a, b\rangle$ as virtual, otherwise as non-virtual. In case the programmer made a wrong prediction, then such a future diamond will have a repeated inheritance semantics (which is undesired most of the time). As mentioned in Section 3.2, we call such layout

algorithm *prediction-guided*, since it is based on user predictions that are assumed never to be falsified.

In order to present our prediction-guided layout, we make the following definitions. Let E be the set of inheritance edges, and $E' \subseteq E$ is a set of predictions (or virtual edges in C++ terminology). Then, we require as in (3) that these predictions are true, i.e., that for every two edges $\langle a, b\rangle, \langle a, c\rangle \in E \setminus E'$ then b and c never share a common descendant d.

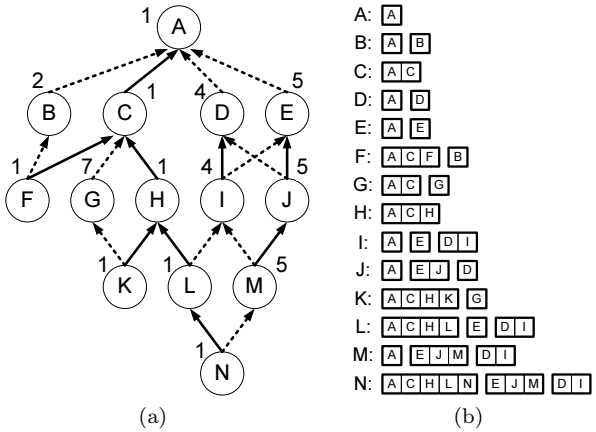
Algorithm 3 computes the two-dimensional address $\langle \phi(t), \rho(t)\rangle$ of a type t, assuming the addresses of all its supertypes have been calculated previously. We assume the existence of a function *typeid* that gives a unique type-id for each type.

Algorithm 3 Prediction-guided Layout Algorithm

Given a set of inheritance edges E, a set of predictions $E' \subseteq E$ satisfying (3), a function *typeid* mapping types to *unique* type-ids, a type t, and the addresses of its supertypes $t' \succeq t$, compute the address of t, i.e., its semi-layer $\phi(t)$ and offset within that semi-layer $\rho(t)$.

1: **If** Exists t' such that $\langle t', t\rangle \in E \setminus E'$ **then** // *t has a parent in* $E \setminus E'$
2: $\quad \phi(t) \leftarrow \phi(t')$; $\rho(t) \leftarrow \rho(t') + 1$
3: **else**
4: $\quad \phi(t) \leftarrow typeid(t)$; $\rho(t) \leftarrow 1$
5: **fi**

Theorem 7.2 will prove that Algorithm 3 computes a layout which is collision-free and contiguous. Figure 6a and Figure 6b depicts the two dimensional layout as computed by Algorithm 3 where edges from the set of predictions E' are dashed, e.g., $\langle \mathsf{A}, \mathsf{B}\rangle \in E'$. Note that the set of predictions E' satisfies (3).



(a) (b)

Fig. 6. (a) The semi-layers computed by Algorithm 3 where edges from the set of predictions E' are dashed, and (b) the resulting two dimensional layout.

Consider, for example, the run of Algorithm 3 on type D in Figure 6a. Since $\langle \mathsf{A}, \mathsf{D}\rangle \in E'$, we have from line 4 that $\phi(\mathsf{D}) = typeid(\mathsf{D}) = 4$ and $\rho(\mathsf{D}) = 1$. Now con-

sider type I. Since $\langle \mathsf{D}, \mathsf{I} \rangle \in E \setminus E'$, we have from line 2 that $\phi(\mathsf{I}) = \phi(\mathsf{D}) = 4$ and $\rho(\mathsf{I}) = \rho(\mathsf{D}) + 1 = 2$.

Observe that Figure 6b is identical to Figure 5c, even though the number of semi-layers in Figure 6a is 5 (semi-layers 1,2,4,5, and 7) whereas in Figure 5a it is 3 (semi-layers 1,2, and 3). In other words, the layout calculated by Algorithm 3 and Algorithm 2 are isomorphic in the sense that there is a mapping m from the semi-layers of Algorithm 3 to those of Algorithm 2 that produces an identical layout. That mapping m is defined as $m(\mathit{typeid}(t)) = \phi(t)$, e.g., $m(7) = m(\mathit{typeid}(\mathsf{G})) = \phi(\mathsf{G}) = 2$.

It was no coincidence that the layouts are isomorphic, and in fact with a specific choice of a predictions set E' Algorithm 3 will always produce an isomorphic layout. In fact, given *any* contiguous and collision-free layout $\langle \phi, \rho \rangle$, we can chose a predictions set E' that produces an isomorphic layout:

$$\text{An edge } \langle a, b \rangle \in E \text{ will be in } E' \text{ if and only if } \phi(a) \neq \phi(b). \tag{4}$$

It is trivial to check that such a definition of E' and the above definition of the mapping m produces an identical layout $\langle \phi, \rho \rangle$.

The prediction set in Figure 6a was selected using (4) with the coloring of Figure 5a. For example, Figure 5a assigns the semi-layers $\phi(\mathsf{A}) = 1$ and $\phi(\mathsf{B}) = 2$, thus we have that $\langle \mathsf{A}, \mathsf{B} \rangle \in E'$ in Figure 6a since $\phi(\mathsf{A}) \neq \phi(\mathsf{B})$.

We finally note that the same set of predictions E' can be used both in the aggressive-inline algorithm of C++ (Section 3.2) and in our two-dimensional algorithm, which gives us a way of comparing the two technique using the same terminology. Consider a type c, and suppose that there exists two edges $\langle a, c \rangle, \langle b, c \rangle \in E \setminus E'$. In the aggressive-inline layout, type c will have a layer that concatenates the layout of a and b. In the two-dimensional layout, type c will share the (semi-)layer of either a or b. Thus, the aggressive-inline layout always assigns an equal or smaller number of layers per object compared to the two-dimensional layout. However, this benefit comes at the cost of dynamic memory overhead and the need for **`this`**-adjustment.

7. CORRECTNESS AND COMPLEXITY ANALYSIS

Having described algorithms for computing the two-dimensional layout, both using whole program information and in separate compilation, we can now proceed to proving their correctness and analyzing their runtime complexity. This section ends with a proof that finding such layout that minimizes the number of semi-layers is an NP-complete problem.

Let $\mathcal{T}$, the input hierarchy, be fixed and arbitrary.

THEOREM 7.1. *Algorithm 2 produces a contiguous and collision-free two dimensional layout.*

PROOF. Let $t \in \mathcal{T}$ be arbitrary, and let $L(t)$ be its entire layout, i.e., the multi-set of all two-dimensional addresses of its ancestors.

(1) By Definition 4.3, the layout is collision-free if all addresses in $L(t)$ are distinct. Let $t_1, t_2 \in \text{ancestors}(t)$, $t_1 \neq t_2$. Then, we need to show that the two-dimensional addresses of t_1 and t_2 are distinct, i.e., either $\phi(t_1) \neq \phi(t_2)$ or $\rho(t_1) \neq \rho(t_2)$.

If t_1 and t_2 are incomparable, then they are in conflict since they share t as a common descendant. Therefore, they are neighbors in the conflict graph, and the coloring heuristic ensured that $\phi(t_1) \neq \phi(t_2)$.

Otherwise, t_1 and t_2 are comparable. If they are in different semi-layers, then we are done. Else, suppose, without loss of generality, that $t_1 \preceq t_2$. Since $t_1 \neq t_2$, we have ancestors$(t_1) \supset$ ancestors(t_2), and the code in lines 3–8 of Algorithm 2 ensures that $\rho(t_1) > \rho(t_2)$.

(2) By Definition 4.4, the layout is contiguous if all addresses in $L(t)$ belonging to a certain semi-layer are consecutive, but this is obvious from the fact that each type is laid out by Algorithm 2 in its respective semi-layer in the position which follows immediately that of all of its proper ancestors in this semi-layer. □

THEOREM 7.2. *Algorithm 3 produces a contiguous and collision-free two dimensional layout.*

PROOF. By the induction on the hierarchy structure. It is trivial to see that $L(r)$, the layout of the root, is collision-free and contiguous. Consider a type t. By induction, $L(t')$ is collision-free and contiguous for all $t' \succeq t$, $t' \neq t$.

Assume by contradiction that $L(t)$ is not collision-free, i.e., there exists two distinct types t and b with a common with the same address, i.e., $\phi(a) = \phi(b)$ and $\rho(a) = \rho(b)$. Let c be the type whose type-id is $\phi(a)$, i.e., $typeid(c) = \phi(a)$. Since this id is unique, then from line 2 in Algorithm 3, there must be a chain of edges in $E \setminus E'$ starting from c and ending in a, i.e.,

$$\langle c, a_1\rangle, \langle a_1, a_2\rangle, \ldots, \langle a_{k-1}, a_k\rangle \in E \setminus E' \text{ and } a = a_k.$$

Similarly, there is such a chain ending in b, i.e.,

$$\langle c, b_1\rangle, \langle b_1, b_2\rangle, \ldots, \langle b_{l-1}, b_l\rangle \in E \setminus E' \text{ and } b = b_l.$$

Let i be the minimal integer for which $a_i \neq b_i$ (there must be such i since $a \neq b$). Note that $\langle a_{i-1}, a_i\rangle, \langle a_{i-1}, b_i\rangle \in E \setminus E'$ and we also have that a_i and b_i have a common descendant t, which contradicts property (3).

Assume by contradiction that $L(t)$ is not contiguous, i.e., there exists a semi-layer $typeid(a)$ which is not contiguous. (Note that from line 4 and uniqueness of type-id we have that $\phi(a) = typeid(a)$.) If $t = a$ then that semi-layer contains just type t since type-id is unique, and from line 4 we have $\rho(t) = 1$ thus this semi-layer must be contiguous. Thus $t \neq a$. If $\phi(t) \neq \phi(a)$ then the layout of the semi-layer $typeid(a)$ is the same as it was in the ancestors of t, contradicting that $L(t')$ is contiguous for all $t' \succeq t$, $t' \neq t$. If $\phi(t) = \phi(a)$ then by line 2 there exists $t' \succeq t$, $\langle t', t\rangle \in E \setminus E'$, and we have that $\rho(t) = \rho(t') + 1$. Since the layout of t' is contiguous, we have that the layout of the semi-layer $typeid(a)$ in t must be contiguous as well. □

Having proved correctness, we now turn to the runtime complexity of the algorithms. It is trivial to see that the runtime complexity of Algorithm 3 is linear. However, the complexity of Algorithm 2 is cubic since it does not rely on the programmer predictions and thus must calculate the conflict graph and solve a coloring problem.

THEOREM 7.3. *Let $n = |\mathcal{T}|$. Then, the runtime complexity of Algorithm 2 is $O(n^3)$.*

PROOF. Lines 2–3 in Algorithm 1, which require constant time, may be executed in certain hierarchies over a fixed fraction of all possible type triplets $\langle t, p_1, p_2 \rangle \in \mathcal{T}^3$, hence the cubic complexity of this algorithm.

Algorithm 2 is time bounded by the simple greedy graph-coloring heuristic whose complexity is $O(n^2)$. The runtime of the inner loop (lines 3–10) of this algorithm is also at most quadratic. □

We now turn to characterizing the problem of finding a contiguous and collision-free two-dimensional layout for a given hierarchy.

LEMMA 7.4. *Let $\langle \phi, \rho \rangle$ be a contiguous and collision-free two-dimensional layout. Then function ϕ is a node coloring of the conflict-graph.*

PROOF. Let t_1 and t_2 be two conflicting types. We need to show that the colors assigned to t_1 and t_2 are distinct, i.e., $\phi(t_1) \neq \phi(t_2)$. Assume to the contrary that t_1 and t_2 are in the same semi-layer $s = \phi(t_1) = \phi(t_2)$.

Since t_1 and t_2 are conflicting they necessarily have at least one common descendant. Let t_3 denote one such descendant. The addresses of t_1 and t_2 are distinct since the layout of t_3 is collision-free. However, since they are in the same semi-layer, they must occupy different positions, i.e., $\rho(t_1) \neq \rho(t_2)$. Without loss of generality, $\rho(t_1) < \rho(t_2)$.

Consider now the type $t_1' \in \text{ancestors}(t_2)$ which is located in position $\rho(t_1)$ in semi-layer s in the layout of t_2. There must be such a type since the layout of t_2 is contiguous. Also, $t_1 \neq t_1'$. (The reason is that t_1 and t_2 are incomparable and therefore t_1 is not present in the layout of t_2.)

We argue that t_1 and t_1' collide in the layout of t_3. Since $t_1' \in \text{ancestors}(t_2) \subset \text{ancestors}(t_3)$ it follows that t_1' is present in the layout of t_3. However, the two-dimensional addresses of t_1 and t_1' are identical. □

The following lemma shows that graph coloring of the conflict graph and finding a contiguous and collision-free two-dimensional layout are equivalent:

LEMMA 7.5. *The conflict graph of $\mathcal{T}$ is Φ-colorable if and only if there is a contiguous and collision-free two-dimensional layout which uses at most Φ semi-layers.*

PROOF. From Theorem 7.1 we know that if the graph is Φ-colorable then there exists such a layout. From Lemma 7.4 we know that if there exists such a layout then the graph is Φ-colorable. □

THEOREM 7.6. *The problem of determining whether a hierarchy has a contiguous and collision-free two-dimensional layout which uses Φ semi-layers is NP-complete for all $\Phi > 2$.*

PROOF. By reduction from graph-coloring. Given a graph $G = (V, E)$, we will build a hierarchy $\mathcal{T}_G$ whose conflict graph has the same chromatic number as G, which proves the theorem using Lemma 7.5.

The construction of $\mathcal{T}_G$ is rather straightforward. There is a type t_v for each $v \in V$, and a type $t_{(u,v)}$ for each $(u, v) \in E$. Also, for all $(u, v) \in E$, type $t_{(u,v)}$ inherits from both t_u and t_v.

It is easy to see that the conflict graph of $\mathcal{T}_G$ has no edges between types $t_{(u,v)}$, and that its conflict graph (ignoring the singleton components $t_{(u,v)}$) is isomorphic to G. □

8. ADDING BI-DIRECTIONALITY TO A TWO-DIMENSIONAL LAYOUT

This section presents three different strategies for realizing a collision-free and contiguous two-dimensional layout in computer memory: the simple and not so efficient *canonical* layout, which is included for the purpose of illustration, the general purpose *compact* layout, which we expect to be used in most cases, and the highly-optimized *inlined* layout which is applicable in some special cases.

All three strategies assume that semi-layers are joined in pairs to form bi-directional layers. Let $\Gamma = \lceil \Phi/2 \rceil$ denote the total number of such layers.

8.1 The Canonical Layout

In the *canonical* layout each object is represented as a pointer to a *Layers Dispatch Table* (LDT) of size Γ. Entry i, $i = 1, \ldots, \Gamma$, of the LDT points to the i^{th} layer of the object.

The canonical layout for the case $\Gamma = 5$ is demonstrated in Figure 7(a). The object depicted in the figure represented by a pointer p to its LDT, which stores pointers to layers L_1, L_3, and L_4. The type of the object is such that it has no fields from the second and the fifth layers. Hence the corresponding entries of the LDT are null.

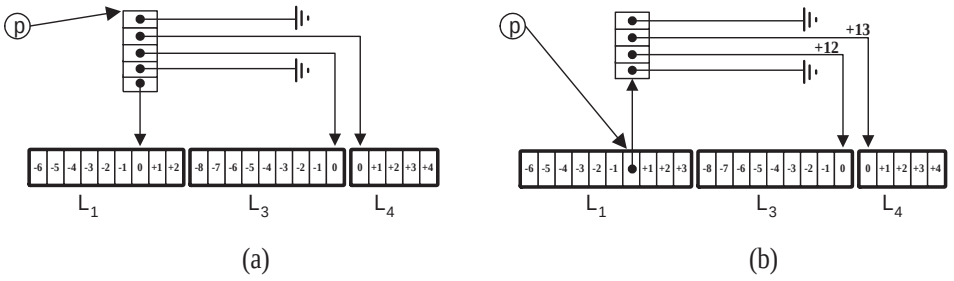


Fig. 7. The canonical (a) and the compact (b) two-dimensional bi-directional layout of an object from a 5-layer hierarchy. Layers L_2 and L_5 are empty in the depicted object.

In general, layers are two directional, and may store fields with both negative and positive offsets. Such is layer L_1 in the figure, with offsets in the range $-6, \ldots, +2$. However, the type of the object depicted has no fields with positive offsets in layer L_3. Similarly, layer L_4 has no fields with negative offsets.

We can see in the figure that each of the layers is contiguous. By placing the layers and the LDT next to each other we obtain a contiguous object layout. The pointers from the LDT to the layers can then be stored as relative offsets.

In realizing a two-dimensional layout in memory we must take care of the fact that different objects have different number of fields, and that some semi-layers take positive directionality while others take negative directionality. Formally, we say a

type t has a *bi-directional two-dimensional address* $\langle \ell(t), \Delta(t) \rangle$, where $1 \leq \ell(t) \leq \Gamma$ is the layer of t and $\Delta(t)$ is the offset of t within its layer (which can be negative). Algorithm 4 shows how these are computed from uni-directional two-dimensional addresses $\langle \phi(t), \rho(t) \rangle$.

Algorithm 4 Produce the canonical two-dimensional bi-directional layout of a hierarchy

Given a two-dimensional layout $\langle \phi, \rho \rangle$ of a hierarchy of types $\mathcal{T}$, compute the number of layers Γ, and for each type t in hierarchy its layer $\ell(t)$ and offset $\Delta(t)$.

1: Let $\Phi = \max_{t \in \mathcal{T}} \phi(t)$
2: Let $\Gamma = \lceil \Phi/2 \rceil$
3: **For all** $t \in \mathcal{T}$ **do** // *Compute the offset and the layer of* t
4: $\quad \Delta(t) \leftarrow 0$ // *Compute the total size of proper ancestors in the same semi-layer as* t
5: $\quad$ **For all** $t' \in \text{ancestors}(t)$, $t' \neq t$ **do**
6: $\quad\quad$ **If** $\phi(t') = \phi(t)$ **then** // *Ancestor* t' *is in the same semi-layer as* t
7: $\quad\quad\quad \Delta(t) \leftarrow \Delta(t) + |t'|$
8: $\quad\quad$ **fi**
9: $\quad$ **od**
10: $\quad \ell(t) \leftarrow \lceil \phi(t)/2 \rceil$ // *Layer* l *hosts semi-layers* $2l-1$ *and* $2l$
11: $\quad$ **If** $\phi(t) \bmod 2 = 0$ **then** // *Even semi-layers are laid out in the negative direction*
12: $\quad\quad \Delta(t) \leftarrow -\Delta(t) - 1$ // *Negative offsets start at* -1
13: $\quad$ **fi**
14: **od**

The main loop of the algorithm computes the layer of each type t, and its offset (which may be negative) within this layer. Lines 4–9 compute the total size of types which precede t in its semi-layer. After computing the layer number (line 10) we turn to making the necessary corrections to the offset. In general, positive semi-layers use offsets $0, +1, +2, \ldots$, while negative semi-layers use offsets $-1, -2, \ldots$ (lines 11–12).

A compiler algorithm for producing the runtime access code in the canonical layout is presented in Algorithm 5. The algorithm takes as input a field `f` and an object pointer `p`, and produces the runtime code for accessing `f` via `p`. The algorithm produces pseudo-C++ runtime code using the **Output** function. To distinguish between compile- and run-time expressions, we use quotation marks and the concatenation operator ",". For example, assuming that $\ell(t) = 1$ then the following code

Output
"`int *x = p[`" , $\ell(t) - 1$, "`];`"

will produce the output `int *x = p[0];`.

Take note that the type t, the layer $\ell(t)$, and the offsets $\Delta(t)$ and $\Delta(\texttt{f})$ are computed at compile time. Moreover, the last statement produces two *runtime constants*: $\ell(t) - 1$, which is the correct index of the pointer to the layer in the

Algorithm 5 An algorithm for generating field access code in the canonical layout

Given `f`, a name of a field of type `int`, and a pointer `p` to an object which uses the *canonical layout*, generate the code sequence (using pseudo-C++ notation) for accessing field `f` in `p`.

1: Let t be the type in which `f` was defined
2: Let $\ell(t)$ be the unique layer of t // $\ell(t)$ *is a positive integer*
3: Let integer $\Delta(t)$ be the offset of t
4: Let $\Delta($`f`$)$ be the offset of `f` within its type // $\Delta($`f`$)$ *is a non-negative integer*
5: **Output**
"`int *layer_ptr = ((int **)p)[`" , $\ell(t) - 1$, "`];`" // *Fetch LDT entry*
"`int &r = layer_ptr[`" , $\Delta(t) + \Delta($`f`$)$, "`];`"

LDT, and $\Delta(t) + \Delta($`f`$)$, which is the index of field `f` within that field. Therefore, a *single* memory dereference is required to compute the field *address*.

8.2 The Compact Layout

It is important to notice that the occupied entries in each layer depend only on the object *type*. Therefore, an offset-based LDT is identical in all objects of the same type and can be shared. The *compact* version of object layout is obtained by employing this sharing and by letting the object pointers reference the first layer directly, which tends to be the largest in our algorithm for assigning fields to layers.

Figure 7b gives an example of the compact layout of the same object of Figure 7a. In the figure we see the same three non-empty layers: L_1, L_3 and L_4. However, the object pointer p now points to offset 0 in layer L_1. At this offset we find the *object type identifier*, which is a pointer to the shared LDT. Notice that the size of layer L_1 was increased by one to accommodate the object type identifier. Also, there are now only four entries in the LDT, which correspond to layers $L_2, \ldots, L_5$.

Algorithm 6 is run by the compiler to generate the code sequence for accessing a field in the compact layout.

If the compiler determines that the field is in the first layer, then the field can be accessed directly—*no* memory dereferences are required for computing its address. If the field however falls in any other layer, then memory must be derefenced once to find the LDT, and then again to find the layer offset. Also, in this case, the addressing mode for the final field access is slightly more complicated since it must add compile- and runtime- offsets.

8.3 The Inlined Layout

The LDT in the example of Figure 7 includes only four entries, all of which are byte-size integers (assuming of course that the object size is less than 256 bytes). The entire LDT can be represented as a single 32 bit word. The *inlined* layout is obtained from the compact layout by inlining the LDT into the object's first layer. At the cost of increasing object space, inlining saves a level of indirection in fetching LDT entries. Note that even if the LDT is stored inside the object, each object must include at least one type identifier for purposes such as subtyping tests and dispatching. Therefore, even in this simple example, the inlined layout uses more space than the compact layout.

Algorithm 6 An algorithm for generating field access code in the compact layout

Given `f`, a name of a field of type **`int`**, and a pointer `p` to an object which uses the *compact layout*, generate the code sequence (using pseudo-C++ notation) for accessing field `f` in `p`.

1: Let t be the type in which `f` was defined
2: Let $\ell(t)$ be the unique layer of t // $\ell(t)$ *is a positive integer*
3: Let integer $\Delta(t)$ be the offset of t
4: Let $\Delta($`f`$)$ be the offset of `f` within its type // $\Delta($`f`$)$ *is a non-negative integer*
5: **If** $\ell(t) = 1$ **then** // *The first layer receives a special treatment*
6: **If** $\Delta(t) \geq 0$ **then** // *If in the positive direction, skip over the LDT pointer*
7: **Output**
"`int &r = ((int *)p)[`" , $\Delta(t) + \Delta($`f`$) + 1$, "`];`"
8: **else** // *The negative direction starts as usual at offset* -1
9: **Output**
"`int &r = ((int *)p)[`" , $\Delta(t) + \Delta($`f`$)$, "`];`"
10: **fi**
11: **else** // *All other layers*
12: **Output**
"`int *p1 = ((int **)p)[0];`" // *Find the address of the LDT*
"`int layer_offset = p1[`" , $\ell(t) - 2$, "`];`" // *Fetch the layer's offset*
"`int &r = p[layer_offset + `" , $\Delta(t) + \Delta($`f`$)$, "`];`"
13: **fi**

We now return to explain the reasons for wishing to minimize the number of layers. One reason for focusing on this objective is that the memory required for LDTs is $\Gamma - 1 \times n$, where $n = |\mathcal{T}|$. LDTs are a source for static memory overhead in the compact layout, and dynamic memory overhead in the inlined layout. More importantly, fewer layers reduce the *likelihood of LDT fetches*, or in other words, the inefficiency of field access. If the number of layers is one, then all fields can be retrieved without any dereferences. Also, if the number of layers is small, then an optimizing compiler might be able to pre-fetch and reuse layer addresses to expedite field access. A probabilistic model of field access is presented in Section 10.2.

8.4 An Example of the Compact Layout

Figure 8 shows the details of the *compact layout* of the running example (the type hierarchy of Figure 4a). Recall that in the compact layout the first layer contains the type-id at offset 0, which is depicted using a dot. The object's pointer will always point to that memory cell.

This layout uses in total two layers and three semi-layers. The first layer has at offset 0 the type-id and both positive and negative semi-layers. The second layer uses only a positive semi-layer.

The figure dedicates a row to portray the the layout of each type in the hierarchy. Each such row depicts, from left to right, three semi-layers: SL_2 (the negative portion of the first layer), SL_1 (the positive portion of the first layer), and SL_3 (the positive portion of the second layer).

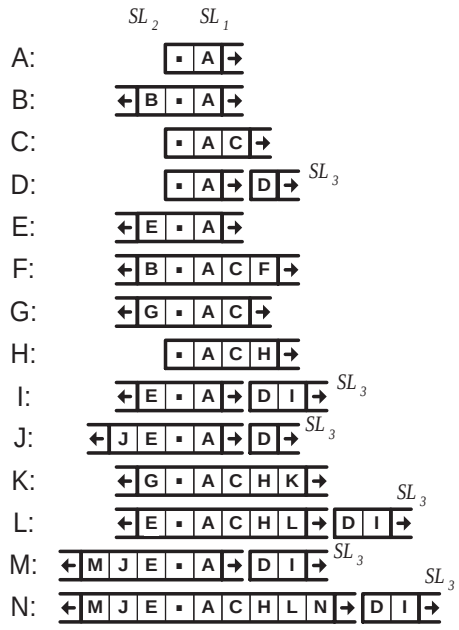


Fig. 8. The two-dimensional bi-directional object layout of the hierarchy in Figure 4a.

For example, the layout of type I, including its three proper ancestors (E, A, and D), is as follows:

—The first layer which contains (i) the fields of E in SL_2 (ii) the type identifier, at the 0^{th} offset, denoted by a small black box in the figure, and (iii) the fields of A in SL_1.

—The second layer, in which semi-layer SL_3 includes the fields of D, and the fields introduced by I itself.

Not all types use all semi-layers: H uses only SL_1, B does not use SL_3, and D does not use SL_2. The type identifier (or, alternatively, the root of the hierarchy) is present in all types; since it is in SL_1, there are no types which do not use this semi-layer.

Notice also the following points:

(1) Semi-layers SL_1 and SL_2, which comprise the first layer, are in a fixed offset; SL_3 occurs at different offsets in different types.

(2) Each type is always placed in the same position in its semi-layer. In other words, the *two-dimensional address* of a type is fixed in all of its descendants. For example, type I is located in the second position in SL_3 in the layouts of all of its descendants: I, L, M, and N. Thus, the two dimensional address of I is $\langle 3, 2\rangle$. Also, the addresses of its (proper) ancestors: E, A, and D, are (respectively), $\langle 2, 1\rangle$, $\langle 1, 1\rangle$, and $\langle 3, 1\rangle$.

(3) The layout is contiguous (Definition 4.4): there are no holes in the layout of any of types A, . . . , N.

For example, the *entire layout* (Definition 4.2) of type I is given by

$$L(\mathsf{I}) = \{\langle 2,1\rangle, \langle 1,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle\}.$$

We can check that set $L(\mathsf{I})$ represents a contiguous layout by verifying that each of the semi-layers is contiguous: no positions other than the first in SL_2 and SL_1, and the first and second in SL_3 are occupied.

(4) The same position in the same semi-layer can be used for different types. For example, the first position of SL_2 stores the fields of B in the layout of B and F, the fields of G in the layout of G and K, and the fields of E in the layout of e.g., J. This does not pose a problem, since at most one of B, G and E is laid out in any type of the hierarchy. Thus, the layout is collision-free (Definition 4.3).

(5) Types are allocated to semi-layers in descending subtyping order. For example, we see that types A, C, H, L and N are placed in this order in SL_1 in the layout of N and that $\mathsf{A} \succeq \mathsf{C} \succeq \mathsf{H} \succeq \mathsf{L} \succeq \mathsf{N}$.

We finally note that the differences between the compact layout as depicted in Figure 8 and the other two layouts (canonical and inlined) are as follows: In the inlined layout, in addition to the type-id that is depicted as a dot, we would also store the offset of the third layer (instead of storing it in the shared type-record). In the canonical layout, we treat all layers in a uniform way, thus the object's pointer would point to an array of pointers to layers as depicted in Figure 7.

9. DATA SET

For the purpose of evaluating the multi-layer object layout scheme, we used an ensemble of 28 type hierarchies, drawn from eight different programming languages, and spanning almost 50,000 types. The first 27 hierarchies[3] were used in our previous benchmarks. A detailed description of their origin, respective programming language, and many of their statistical and topological properties can be found elsewhere [Zibin and Gil 2001; 2002]. (Even though multiple inheritance of fields is not possible in JAVA, the JAVA hierarchies are still useful in characterizing how programmers tend to use multiple inheritance. When translating these JAVA hierarchies to C++ one would get multiple inheritance hierarchies. Although these hierarchies are degenerate in the sense that no fields are defined in classes that correspond to interfaces, we assumed that a programmer might add some field definitions and method code.) To these we added FLAVORS [Moon 1986], a 67-type hierarchy representing the *multi-inheritance core* of the FLAVORS language benchmark used by Pugh and Weddell [1990, Fig. 5].

Together, the hierarchies span a range of sizes, from 67 types (in IDL and FLAVORS) up to 8,793 types in MI: IBM SF, the median being 930 types. The hierarchies are relatively shallow, with heights between 9 and 17. Most types have just one parent, and the overall average number of parents is 1.2. In these and other respects, the hierarchies are not very different from balanced binary trees [Eckel and Gil 2000].

[3]IDL, MI: IBM XML, JDK 1.1, Laure, Ed, LOV, Cecil2, Cecil-, Unidraw, Harlequin, MI: Orbacus Test, MI: HotJava, Dylan, Cecil, Geode, MI: Orbacus, Vor3, MI: Corba, JDK 1.18, Self, Vortex3, Eiffel4, MI: Orbix, JDK 1.22, JDK 1.30, MI: JDK 1.3.1, and MI: IBM SF.

The number of ancestors is typically small, averaging less than 10 in most hierarchies. Exceptions are the Geode and the Self hierarchies, which make an extensive use of multiple inheritance. In Geode, there are 14 ancestors on average per type, and there exists a type with as many as 50 ancestors. Self has 31 ancestors in average per type. The topology of Self is quite unique in that almost all types in it inherit from a type with 23 ancestors. Table I below gives (among other information), the number of types in each hierarchy, and the maximal and average number of ancestors.

10. EXPERIMENTAL RESULTS

The next subsection presents the results of computing the two-dimensional layout with Algorithm 1 and Algorithm 2 on our data set, followed by a comparison with previous work.

10.1 Two-dimensional Bi-Directional layout

Since the layout depends on a graph-coloring heuristic (Line 1 of Algorithm 2), we would like first to examine the quality of the coloring computed by our greedy coloring heuristic. We remind the reader that if a graph has a clique of size k, then it cannot be colored by fewer than k colors. Although it is not easy to find cliques in general graphs, some cliques can be efficiently found in conflict graphs. Consider a type t and its set of ancestors ancestors(t). Let $P_t \subseteq$ ancestors(t) be a set of types which are pair-wise incomparable. Then any $t_1, t_2 \in P_t$ are in conflict, and the set P_t is a clique in the conflict graph. Finding a maximal set of incomparable nodes in a hierarchy is a standard procedure of finding a maximal anti-chain in a partial order [Trotter 1992].

Table I compares the number of colors and layers with the predictions of the lower bound thus found.

Let

$$\omega_t = \max\{|P_t| \mid P_t \subseteq \text{ancestors}(t) \text{ is a set of pair-wise incomparable types}\},$$

i.e., ω_t is the size of the maximal anti-chain among the ancestors of t. Then,

$$\omega = \max_{t \in \mathcal{T}} \{\omega_t\}$$

is a lower bound on the number of colors (or semi-layers), and $\lceil \omega/2 \rceil$ is a lower bound on the number of layers Γ. We see in the table that $\Phi > \omega$ only in seven hierarchies: Flavors, Ed, LOV, MI: Orbacus Test, MI: HotJava, Geode and MI: Corba. In these seven cases, $\Phi = \omega + 1$, so the number of colors was off by at most one from the lower bound. Further, as the next two columns indicate, the situation that the number of layers is greater than the prediction of the lower bound, occurs in only three hierarchies: Ed, MI: HotJava and MI: Corba.

It is also interesting to compare the number of colors and the number of layers with the maximal number of ancestors, denoted $\alpha = \max(\theta_t)$. In our data-set, the number of colors is never greater than the maximal number of ancestors, and is typically much smaller than it. The number of entries in the LDT is even smaller, since every two colors are mapped to a single layer.

The maximal number of layers in the field dispatching technique is exactly α, since each layer is a singleton. The field dispatch matrix can be compressed using

Hierarchy $\langle \mathcal{T}, \preceq \rangle$	$n = \lvert\mathcal{T}\rvert$	ω [a]	Φ [b]	$\lceil \omega/2 \rceil$	$\lceil \Phi/2 \rceil$	$\max(\theta_t)$[c]	$\text{avg}(\Gamma_t)$[d]	$\text{avg}(\theta_t)$[e]
FLAVORS	67	3	4	2	2	13	1.6	4.9
IDL	67	2	2	1	1	9	1.0	4.8
MI: IBM XML	145	5	5	3	3	14	1.5	4.4
JDK 1.1	226	2	2	1	1	8	1.0	4.2
Laure	295	3	3	2	2	16	1.1	8.1
Ed	434	12	13	6	7	23	3.2	8.0
LOV	436	13	14	7	7	24	3.5	8.5
Cecil2	472	8	8	4	4	29	2.0	7.4
Cecil-	473	8	8	4	4	29	2.0	7.4
Unidraw	614	3	3	2	2	10	1.0	4.0
Harlequin	666	14	14	7	7	31	1.9	6.7
MI: Orbacus Test	689	3	4	2	2	12	1.3	3.9
MI: HotJava	736	14	15	7	8	23	2.0	5.1
Dylan	925	3	3	2	2	13	1.1	5.5
Cecil	932	6	6	3	3	23	1.7	6.5
Geode	1,318	21	22	11	11	50	5.1	14.0
MI: Orbacus	1,379	11	11	6	6	19	1.6	4.5
Vor3	1,660	6	6	3	3	27	1.6	7.5
MI: Corba	1,699	6	7	3	4	18	1.3	3.9
JDK 1.18	1,704	12	12	6	6	16	1.2	4.3
Self	1,802	24	24	12	12	41	10.7	30.9
Vortex3	1,954	8	8	4	4	30	1.7	7.2
Eiffel4	1,999	15	15	8	8	39	2.2	8.8
MI: Orbix	2,716	6	6	3	3	13	1.1	2.8
JDK 1.22	4,339	14	14	7	7	17	1.5	4.4
JDK 1.30	5,438	15	15	8	8	19	1.5	4.4
MI: JDK 1.3.1	7,401	21	21	11	11	24	1.5	4.4
MI: IBM SF	8,793	13	13	7	7	30	2.3	9.2

[a] the maximal size of an anti-chain in the ancestors of any type $t \in \mathcal{T}$
[b] the number of colors (or semi-layers) used by Algorithm 2
[c] $\max\{\theta_t \mid t \in \mathcal{T}\}$
[d] $\frac{1}{n} \sum_{t \in \mathcal{T}} \Gamma_t$
[e] $\frac{1}{n} \sum_{t \in \mathcal{T}} \theta_t$

Table I. Statistics on the input hierarchies, including the number of colors and layers found by the greedy heuristic compared with the maximal anti-chain lower bound

method dispatching techniques, such as selector coloring [Dixon et al. 1989; Pascal and Royer 1992]. A lower bound on the space requirement of selector coloring is $n \times \alpha$. We therefore have that the static memory of our layout scheme $n \times \Gamma$ is superior to that of the field dispatch matrix compressed using selector coloring. (Using other techniques for compressing the dispatching matrix [Zibin and Gil 2002] might yield a smaller field dispatch matrix.)

The next two columns of Table I give another comparison of hash-table implementation of the LDT with a hash table implementation of the field dispatch matrix. We see that the number of layers which each object uses is typically small. No more than 3.5 in all but the Self and Geode hierarchies. In all hierarchies, we see that the average number of ancestors is much greater than the average number of layers.

This shows that **(i)** the greedy heuristic is successful in compressing multiple types into layers, and consequently that **(ii)** the LDT places weaker demands than the field dispatch matrix on static memory.

10.2 Comparison with Previous Work

Recall the evaluation criteria of Section 1.3: creation time, field access efficiency, dynamic- and static-memory overhead. We next compare our new layout with three C++ layout schemes: the standard, simple-inline, and aggressive-inline [2000]. There is no fair comparison of our layout with Pugh and Weddell [1990] *fixed offsets* algorithm since our layout has zero dynamic memory overhead but with additional cost to field access whereas the fixed offset algorithm has dynamic memory overhead (objects have "holes") but field access requires one load. For example, in the FLAVORS hierarchy Pugh and Weddell reported 6% dynamic memory overhead (assuming a single instance per type). Our scheme uses only two layers for this hierarchy, and the probability that a field access would require extra dereferences is only 0.19.

Creation time: As argued in Theorem 7.3, the theoretical complexity of the algorithm is cubic. However, the fact that the average number of ancestors in actual hierarchies is small, makes its runtime much more tolerable.

By applying some rather straightforward algorithmic optimizations, e.g., considering in line 2 of Algorithm 1 only types which have more than one parent, the run times were reduced even further. On a Pentium III, 900Mhz machine, equipped with 256MB internal memory and running a Windows 2000 operating system, Algorithm 2 required less than 10 mSec in 19 hierarchies. Seven hierarchies required between 10 mSec and 50 mSec. The worst hierarchy was MI: IBM SF which took 400 mSec. The total runtime for all hierarchies was 650 mSec, which gives on average 13μSec of CPU time per type. The runtime of the C++ aggressive-inline procedure on the same hardware is much slower. For example, aggressive inline of MI: IBM SF took 3,586 mSec, i.e., about 9 times slower. Simple inline of MI: IBM SF took 2,294 mSec, which is still much slower.

Field access efficiency: Since the hierarchies were drawn from different languages and were not associated with any application programs, we were unable to directly measure the actual cost of field access in the various layout schemes. Moreover, replacing the layout scheme of C++ is a major design challenge since it requires changing the dispatching mechanism as well due to strong coupling between dispatching and layout in C++ . (The layout of the virtual tables is identical to the way the object's fields are laid out.) We can however derive other metrics to compare the costs of the new layout technique with that of prior art.

For example, the number of layers used by a given type, gives an indication on the number of different dereferences required to access *all* the object fields. The corresponding metric in C++ is the number of virtual bases, which can be accessed only by dereferencing a VBPTR.

Figure 9 compares the average number of layers of the new scheme with that of the standard C++ implementation, the simple inlined implementation and the aggressive inlined implementation.

We see in the figure that with the exception of Self hierarchy (which as we mentioned above has a very unique topology), the new layout scheme is always

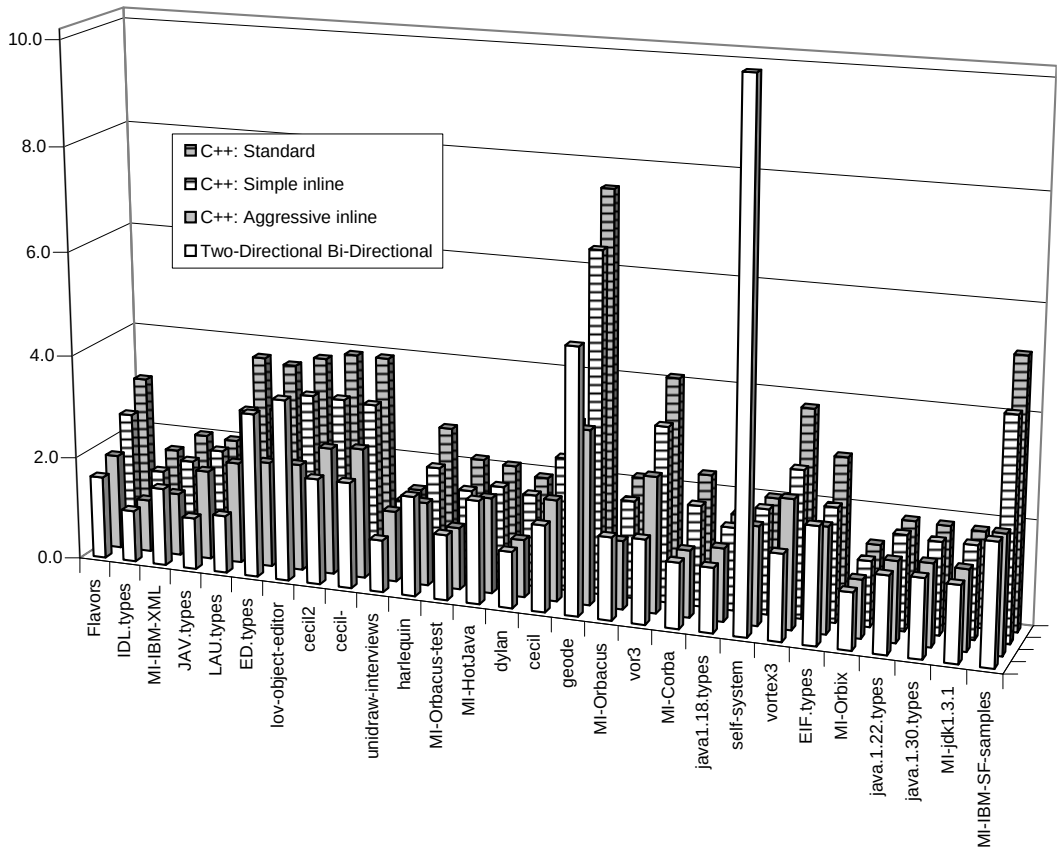


Fig. 9. Average no. of layers in different hierarchies.

superior to the standard- and simple-inlined implementation of C++ . Moreover, the new scheme is superior or comparable with the aggressive-inline layout scheme, with the exception of four hierarchies: Ed, LOV, Geode and Self. Comparing the *maximal*- rather than the *average*- number of layers yields similar results.

We also offer a more sophisticated theoretical model for comparing the performance of various schemes of object layout which involve indirection to access various fields. Suppose that a certain field was retrieved from a certain layer. Then, a good optimizing compiler should be able to reuse the address of this layer in retrieving other fields from this layer. Even in the standard C++ layout, the compiler may be able to reuse the address of a virtual base to fetch additional fields from this base.

For a fixed type t, and for a sequence of k field accesses, we would like to compute $A_t(k)$, the expected number of extra dereferences required to access these fields. Due to missing empirical data from our ensemble of hierarchies, we make two simplifying assumptions:

(1) *Uniform class size.* The number of fields introduced in each type is the same.

Although evidently inaccurate, this assumption should not be crucial to the results. We do expect that most classes introduce a small number of fields, with a relatively small variety.

(2) *Uniform access probability.* The probability of accessing any certain field is fixed, and is independent of the fields accessed previously, nor of the type in which the field is defined. This assumption is clearly in contradiction to the *principle of locality of reference*.
However, as we shall see, locality of reference improves the performance of layout schemes. It is not clear whether this improvement contribute more to any specific scheme.

The θ_t ancestors of t are laid out in Γ_t different layers or virtual bases, such that layer i (virtual base i) has $\theta_t(i)$ ancestors. The first layer can always be accessed directly. Access to a field in layer i in step k requires a dereference operation, if that layer was not accessed in steps $1, \ldots, k-1$.

Let $X_t(i)$, $i = 2, \ldots, \Gamma_t$ be the random binary variable which is 1 if a field of level i was not referenced in any of the steps $1, \ldots, k$. Then,

$$\mathbf{Prob}[X_t(i) = 1] = \mathbf{Exp}(X_t(i)) = \left(1 - \frac{\theta_t(i)}{\theta_t}\right)^k .$$

Additivity of expectation allows us to sum the above over i, obtaining that the expected number of levels (other than the first) which were not referenced is

$$\sum_{i=2}^{\Gamma_t} \left(1 - \frac{\theta_t(i)}{\theta_t}\right)^k .$$

Using the linearity of expectation, we find that the expected number of referenced levels, i.e., the number of dereferences is simply

$$A_t(k) = (\Gamma - 1) - \sum_{i=2}^{\Gamma_t} \left(1 - \frac{\theta_t(i)}{\theta_t}\right)^k . \tag{5}$$

Averaging over an entire type hierarchy, we define

$$A(k) = \frac{1}{n} \sum_{t \in \mathcal{T}} A_t(k) \tag{6}$$

Figure 10 gives a plot of $A(k)$ vs. k in four sample hierarchies (other hierarchies give rise to similar graphs) in the layout schemes field dispatching, standard C++ layout, simple inline (S-Inline), aggressive inline (A-Inline), and our two-dimensional bi-directional layout (TDBD). Values of $A(k)$ were computed using (5) and (6) in the respective hierarchy and object layout scheme. For field dispatching, we set $\theta_t(i) = 1$.

It is interesting to see that in all hierarchies and in all layout schemes, the expected number of dereferences is much smaller than the number of actual fields accessed. It is also not surprising that $A(k)$ increases quickly at first and slowly later. As expected, the new scheme is much better than field dispatching. The graphs give hope of saving about 75% of the dereferences incurred in field dispatching. (Note however that the model does not take into account any optimizations which runtime systems may apply to field dispatching.)

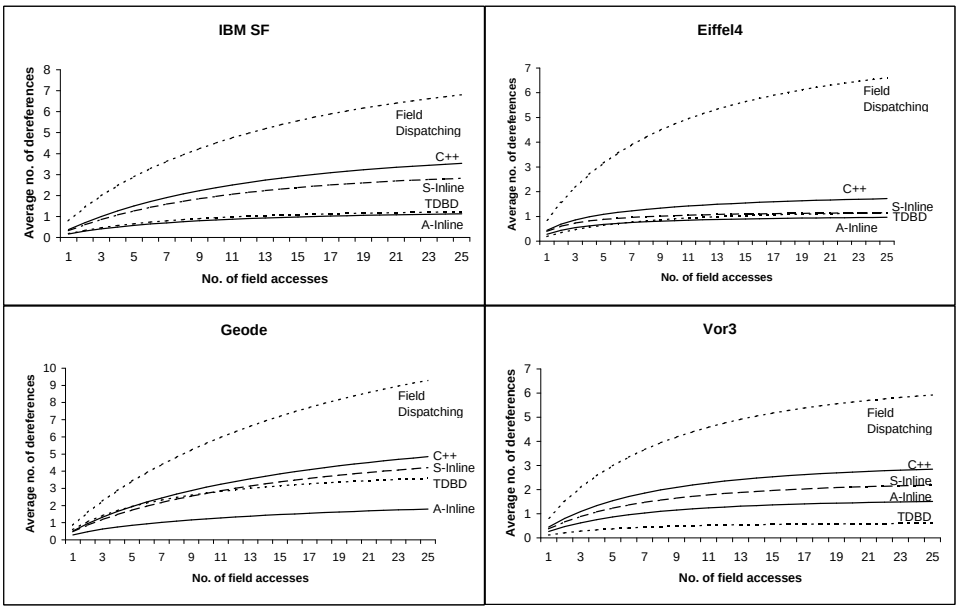


Fig. 10. Average no. of dereferences vs. no. of field accesses in four hierarchies.

The other, C++ specific techniques are also more efficient than field dispatching. We now turn to comparing these with our scheme. In the Vortex3 hierarchy the new scheme dramatically improves the expected number of dereferences compared to any of the C++ layout schemes. The new scheme is also the best in smaller k values in the Eiffel4 hierarchy, and is comparable to aggressive inline with greater values of k. Another typical behavior is demonstrated by MI: IBM SF, in which the new scheme is almost the same as aggressive-inline. In the Geode hierarchy which is one of the two hierarchies in which the two-dimensional bi-directional scheme cannot find a good partitioning into a small number of layers, we find that aggressive inline gives the best results in terms of field access efficiency. Still, even in this hierarchy the new scheme is better than the standard C++ implementation and the simple-inline outline heuristic.

Dynamic memory: As mentioned before, our layout incurs no dynamic memory overhead. In contrast, the various C++ layout schemes sometimes store multiple VPTRs, as explained in Section 3.2. Table II shows the extra dynamic memory consumed by those VPTRs.

Curiously, the four hierarchies in which the new scheme does not perform as well, Ed, LOV, Geode and Self, are exactly the hierarchies in which the C++ schemes, including the highly optimized aggressive inline waste the most amount of dynamic memory.

Static memory: The static memory overhead of our layout can be found in Table I. We see that on the worst case the number of layers was 12, in which case our layout stores 12 bytes per type. For example, in a hierarchy with 10,000 types and 12 layers, the static memory overhead of our technique is 120KB.

Hierarchy	Average			Median			Maximum		
	C++	S-In	A-In	C++	S-In	A-In	C++	S-In	A-In
FLAVORS	3.4	3.2	2.4	3	3	2	9	8	5
IDL	1.9	1.6	1.2	2	2	1	3	2	2
MI: IBM XML	2.8	2.8	2.0	2	2	1	9	9	6
JDK 1.1	2.1	2.0	1.8	2	2	2	4	4	3
Laure	3.9	3.2	2.3	4	3	2	8	7	5
Ed	5.2	5.0	4.2	4	4	4	16	16	12
LOV	5.6	5.5	4.6	5	5	4	17	17	13
Cecil2	4.6	4.4	3.4	3	3	3	17	15	9
Cecil-	4.6	4.3	3.5	3	3	3	17	15	9
Unidraw	1.4	1.4	1.4	1	1	1	4	3	3
Harlequin	3.6	3.2	2.7	2	2	2	21	19	16
MI: Orbacus Test	2.5	2.1	1.7	2	2	1	8	6	5
MI: HotJava	2.9	2.9	2.7	2	2	2	17	17	15
Dylan	2.0	1.9	1.3	2	2	1	7	6	5
Cecil	3.7	3.5	2.7	3	3	2	16	13	8
Geode	9.9	9.5	8.3	9	9	7	32	31	27
MI: Orbacus	2.8	2.6	2.2	2	2	1	13	12	11
Vor3	4.6	4.2	3.5	4	3	3	17	14	11
MI: Corba	2.6	2.3	1.7	2	2	1	14	12	10
JDK 1.18	1.9	1.9	1.7	2	2	1	14	13	12
Self	21.2	21.2	21.1	22	22	22	26	25	25
Vortex3	4.4	3.8	3.4	3	3	3	18	15	11
Eiffel4	3.7	3.4	3.1	2	2	2	20	17	16
MI: Orbix	1.5	1.4	1.3	1	1	1	7	7	6
JDK 1.22	2.4	2.3	2.1	2	2	2	16	15	14
JDK 1.30	2.4	2.3	2.1	2	2	2	17	17	16
MI: JDK 1.3.1	2.3	2.3	2.0	2	2	1	23	22	21
MI: IBM SF	5.8	5.8	3.6	6	6	3	16	16	13
Total	4.2	4.0	3.3	-	-	22	32	31	27
Median	3.2	3.0	2.4	2	2	2	16	14.5	11
Minimum	1.4	1.4	1.2	1	1	1	3	2	2
Maximum	21.2	21.2	21.1	22	22	22	32	31	27

Table II. No. of VPTRs using standard C++ layout, simple inline (S-In), and aggressive inline (A-In)

For the C++ layouts, the VBPTRs can be a source for static memory (if they are not inlined in the object, in which case they contribute to the dynamic memory). A lower bound on the average number of VBPTR in an object is the number of layers in Figure 9. (An object can also have inessential VBPTRs [2000].) We see that in the worst case the average number of VBPTRs per type is less than 8.

11. CONCLUSIONS AND OPEN PROBLEMS

The two-dimensional bi-directional object layout scheme enjoys the following properties: **(i)** the dynamic memory overhead per object is a single type-identifier, **(ii)** the static memory per type is small: at most 11 cells in our data set, but usually only around 5 cells, **(iii)** small time for computing the layout: an average of 13 μSec per type in our data set, and **(iv)** good field access efficiency as predicted

by our analytical model: the new scheme always improves upon the field dispatching scheme and on the standard C++ layout model. The new scheme compares favorably even with a highly optimized C++ layout.

We note that the new scheme does not rely on **`this`**-adjustment, and in the few hierarchies where the aggressive-inline of C++ won, it was with the cost of large dynamic memory overheads, e.g., as much as 21 VPTRs on average per object in the Self hierarchy.

The one-dimensional bi-directional layout of Pugh and Weddell's [1990] realizes field access in a single indirection, but it may leave holes in some objects. In comparison, our two-dimensional bi-directional layout has no dynamic memory overheads, but a field access might require extra dereferences. In the FLAVORS hierarchy Pugh and Weddell reported 6% dynamic memory overhead (assuming a single instance per type). Our scheme uses only two layers for this hierarchy, and the probability that a field access would require extra dereferences is only 0.19.

Directions for future work include empirical study of frequencies of field accesses, and further reducing the static memory overheads. In dynamically typed languages where fields can be overloaded, the layout algorithm must color fields instead of types. Empirical data should be gathered to evaluate the efficiency of the layout algorithm in such languages.

REFERENCES

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, Reading, Massachusetts.

BORNING, A. H. AND INGALLS, D. H. H. 1982. Multiple inheritance in Smalltalk80. In *Proc. of the Second National Conference on Artificial Intelligence*. MIT Press, Pittsburgh, 234–238.

CARGILL, T. A., COX, B., COOK, W. R., LOOMIS, M., AND SNYDER, A. 1993. Is multiple inheritance essential to OOP? Panel discussion at the (OOPSLA'95) (Washington, DC).

CHAMBERS, C. 1993. The Cecil language, specification and rationale. Tech. Rep. TR-93-03-05, University of Washington, Seattle.

DIXON, R., MCKEE, T., VAUGHAN, M., AND SCHWEIZER, P. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. of the Fourth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89)*, N. K. Meyrowitz, Ed. ACM SIGPLAN Notices 24(10), New Orleans, Louisiana, 211–214.

ECKEL, N. AND GIL, J. 2000. Empirical study of object-layout strategies and optimization techniques. In *Proc. of the Fourteenth European Conference on Object-Oriented Programming (ECOOP'00)*, E. Bertino, Ed. Lecture Notes in Computer Science, vol. 1850. Springer Verlag, Sophia Antipolis and Cannes, France, 394–421.

GIL, J. AND SWEENEY, P. F. 1999. Space- and time-efficient memory layout for multiple inheritance. In *Proc. of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*. ACM SIGPLAN Notices 34 (10), Denver, Colorado, 256–275.

GOLDBERG, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, Massachusetts.

LIPPMAN, S. B. 1996. *Inside The C++ Object Model*, Second ed. Addison-Wesley Publishing Company, Reading, Massachusetts.

MAGNUSSUN, B., MEYER, B., AND ET AL. 1994. Who needs need multiple inheritance. In *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'94 Europe)*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, Versailles, France.

MOON, D. A. 1986. Object-oriented programming with flavors. In *Proc. of the First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, N. K. Meyrowitz, Ed. ACM SIGPLAN Notices 21(11), Portland, Oregon, USA, 1–8.

Myers, A. C. 1995. Bidirectional object layout for separate compilation. In *Proc. of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*. ACM SIGPLAN Notices 30(10), Austin, Texas, USA, 124–139.

Pascal, A. and Royer, J. 1992. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proc. of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, A. Paepcke, Ed. ACM SIGPLAN Notices 27(10), Vancouver, British Columbia, Canada, 110–126.

Pugh, W. and Weddell, G. 1990. Two-directional record layout for multiple inheritance. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'94)*. ACM SIGPLAN Notices 25(6), White Plains, New York, 85–91.

Pugh, W. and Weddell, G. 1993. On object layout for multiple inheritance. Technical Report CS-93-22, University of Waterloo—Department of Computer Science. May.

Shalit, A. 1997. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Stroustrup, B. 1994. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Stroustrup, B. 1997. *The C++ Programming Language*, Third ed. Addison-Wesley Publishing Company, Reading, Massachusetts.

Sweeney, P. F. and Burke, M. 2003. Quantifying and evaluating the space overhead for alternative c++ memory layouts. *Software - Practice and Experience 33,* 7, 595–636.

Trotter, W. T. 1992. *Combinatorics and Partially Ordered Sets: Dimension Theory*. John Hopkins University Press, Baltimore, MD.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proc. of the Twelfth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*. ACM SIGPLAN Notices 32(10), Atlanta, Georgia, USA, 125–141.

Zibin, Y. and Gil, J. 2001. Efficient subtyping tests with PQ-encoding. In *Proc. of the Sixteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*. ACM SIGPLAN Notices 36(11), Tampa Bay, Florida, 96–107.

Zibin, Y. and Gil, J. 2002. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proc. of the Seventeenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*. ACM SIGPLAN Notices 37(11), Seattle, Washington, 142–160.

Zibin, Y. and Gil, J. 2003. Incremental algorithms for dispatching in dynamically typed languages. In *Proc. of the Thirtieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, New York, NY, USA, New Orleans, Louisiana, USA, 126–138.

A. NOTATION

Acronym	Expansion
LDT	Layers Dispatch Table
VBPTR	Virtual Base Pointer (in C++)
VPTR	Virtual Function Pointer (a pointer to a virtual function table in C++)

Symbol	Denotes
$\preceq$	The subtype relation
$\mathcal{T}$	The set of types
$\mathsf{A}, \ldots, \mathsf{J}, \mathsf{R}$	Concrete types
a, b, t, t'	Type variables; $a, b, t, t' \in \mathcal{T}$
r	The root of the type hierarchy
$\|t\|$	The number of fields introduced in t
`o`	Some object
`f`	Some field
n	Number of types; $n = \|\mathcal{T}\|$
Φ	The number of colors required to color the conflict graph, which is also the number of semi-layers
Γ	The number of layers; $\Gamma = \lceil \Phi/2 \rceil$
Γ_t	The number of non-empty layers in the layout of type t
$\rho(t)$	The integral position of t in its semi-layer, $\rho(t) > 0$
$\Delta(t)$	The integral offset of t in its layer (can be negative)
$\Delta(\mathtt{f})$	The offset of a field `f` within its type
$\phi(t)$	The semi-layer of type t, $1 \leq \phi(t) \leq \Phi$
$\ell(t)$	The layer of type t, $1 \leq \ell(t) \leq \Gamma$
$\langle \phi(t), \rho(t) \rangle$	The uni-directional two-dimensional address of a type t
$\langle \ell(t), \Delta(t) \rangle$	The bi-directional two-dimensional address of a type t
θ_t	The number of ancestors of type t
$\theta_t(i)$	The number of ancestors of type t in layer i
$L(t)$	The entire layout of a type t, i.e., the multi-set of addresses of t and its ancestors
$A_t(k)$	The expected number of extra dereferences required to access k random fields in t
$A(k)$	The average of $A_t(k)$ over an entire type hierarchy

Received August 2004