Two-Dimensional Bi-directional Object Layout

Yoav Zibin* and Joseph (Yossi) Gil

Technion—Israel Institute of Technology
{zyoav,yogi}@cs.technion.ac.il

Abstract. C++ object layout schemes rely on (sometimes numerous) compiler generated fields. We describe a new 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 the new scheme improves field access efficiency over standard implementations, and competes favorably with (the non-space optimal) highly optimized C++ specific implementations. The benchmark includes a new 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 microseconds per type on a contemporary architecture (Pentium III, 900Mhz, 256MB machine), even in very large hierarchies.

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 [1, 7]. 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., [17, 18, 19] for recent surveys). Another great concern in the design of runtime systems for multiple inheritance hierarchies is efficient object layout. To this end, both general purpose [9] and C++ language specific [5, 4] object layout schemes were previously proposed in the literature.

The various C++ layout schemes are not space-optimal since they introduce (sometimes many) compiler generated fields into the layout. They are also not time-optimal since access to certain fields (in particular, those defined in virtual bases) requires several memory dereferences. This paper revisits the object layout problem in the general, language-independent setting. Our new object layout scheme is space optimal, i.e., objects are contiguous, and contain *no compiler generated fields*. Hence, in terms of space, it is superior to C++ layout schemes. It is also superior in terms of field access efficiency to the space-optimal *field dispatching* technique[1] employed by many object oriented languages.

* Contact author.

[1] In the field dispatching technique we encapsulate fields in accessor methods.

L. Cardelli (Ed.): ECOOP 2003, LNCS 2743, pp. 329–350, 2003.

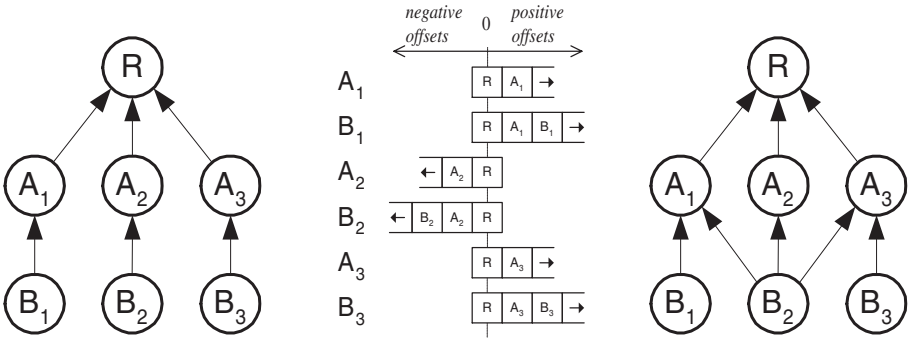

Fig. 1. A small single inheritance hierarchy (a), a possible object layout for this hierarchy (b), and a multiple inheritance hierarchy in which there is no contiguous layout for all objects (c)

We say that the layout is *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 [4]. Even though the new layout is not C++ specific, it competes favorably in this respect with aggressive inlining [4], arguably the best C++ layout scheme.

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

A possible object layout of the types defined in this hierarchy is shown in Figure 1b. The fields of A_1 are laid out just after R. The layout of B_1 adds its own fields in increasing offsets. All types inheriting from A_1 and B_1 will have positive directionality. Types A_2 and B_2 are laid out in negative offsets. This should also be the directionality of any of their descendants. Types A_3 and B_3 and all of their descendants 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 either in negative or 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 from B_2 to A_1 and A_3. 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 wrote: "repeated inheritance is an abomination".[2]

With the addition of multiple inheritance, a layout for 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 the strong conformance requirement (or fixed offsets [9]) which we implicitly made:

> **The Strong Conformance Requirement:** Every type must be laid out in the same offset in all of its descendants.

If the layout of A_1, A_2 and A_3 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.

Thus, we see that it is sometimes impossible to maintain the strong conformance requirement and contiguous object layout. Our new scheme resolves the conflict by sacrificing the strong conformance requirement. In particular, each object is laid out in one or more layers, where each layer uses a bidirectional layout. The above difficulty is removed by placing (say) type A_3 in a different layer.

We note that separate compilation discovers too late that two base types compete for the same memory location, i.e., after the layout of these base types was determined. For this reason, our layout scheme, just as all other optimizing layouts, relies on whole program analysis.

Outline Pertinent definitions are given in Section 2, which also lists some of the standard simplifications of the object layout problem. Section 3 describes the criteria used in evaluating object layout schemes, using these to place our result in the context of previous work. The actual layout, which comes in three versions is described in Section 4. Section 5 presents the algorithm for computing the actual layout. Section 6 describes the data-set used in the benchmark, while Section 7 gives the experimental results. Finally, conclusions and directions for future research are given in Section 8.

Epilog We have recently learnt that similar results were independently obtained by Pugh and Weddell and described in a 1993 technical report [10]. Their work suggests a similar layout algorithm, using fields instead of types, and includes several theoretical bounds on complexity. Our work takes a more empirical slant.

[2] Words of an anonymous reviewer to [5].

2 Definitions

Leading to a more exact specification of the problem, this section makes precise notions such as a hierarchy, incomparable types, and introduced and accessible fields in a type.

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 antisymmetric. Let $a, b \in \mathcal{T}$ be arbitrary. Then, if $a \preceq b$ holds we say that a is a *subtype* of b and that b is a *supertype* of a. 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.

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 $a \in \text{ancestors}(a)$.

Types in a hierarchy may *introduce* fields, which can be thought of as unique names or selectors. We assume that there is no *field overriding*, i.e., that the same field name can only be used once in each type. Although C++ (and other languages) allow a derived class to reuse the name of a `private` field defined in a base class, our assumption is trivially satisfied by simple renaming.

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.

The problem of object layout in dynamically typed languages is not very interesting and excluded from the domain of discourse. In languages such as SMALLTALK, fields access is restricted to the methods of the defining object. With this restriction, the strong conformance requirement does not need to be satisfied[3]: The object layout problem then becomes trivial, even with the face of multiple inheritance. If however a dynamically typed language supports non-private fields, then 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 received extensive coverage in the literature [17, 18, 19].

For simplicity, we assume that all fields are of the same size. For a type $t \in \mathcal{T}$, let $|t|$ denote the number of fields introduced in t. The *accessible* fields of a type include all fields introduced in it and in any of its proper supertypes.

Given a type hierarchy, the *object layout problem* is to design a *layout scheme* for the objects of each of the types in the hierarchy, and a method for accessing at runtime the accessible fields of each type. Specifically, given a field `f` and an object address `o` of type t, the runtime system should be able to compute the

[3] In fact, even the weak conformance requirement (defined later in Section 3) is not satisfied.)

address of `o.f`. The selector `f` is a compile time constant, while `o` is supplied only at runtime.

3 The Object Layout Problem

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 must include 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 a common root type (e.g., type `R` in Figure 1), 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.
   It is common that all fields introduced in a certain type are laid out consecutively. Since `f` is supplied at compile time, the type t' in which `f` was introduced can be precomputed. 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.

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 [9] investigated more efficient layout schemes which still fulfill the strong conformance requirement. The dynamic memory overhead of their main bidirectional object layout scheme is in one case study only 6%, compared to 47% in a unidirectional object layout. The authors also showed that the problem of determining whether an optimal bidirectional layout exists is NP-complete.

At the other extreme stands what may be called *field dispatching* layout scheme, which is employed by many dynamically typed programming languages including Cecil [2] and Dylan [12]. 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, we encapsulate fields in accessor methods. If a field position changes in a subtype, we override its accessor. 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 (see e.g., [18] for a recent survey). A different implementation is found in the SmallEiffel compiler [16], in which a static branch code over the dynamic type of the object finds the required base offset.

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.

An interesting tradeoff between the two extremes is offered by the memory model of C++ [6]. C++ distinguishes between `virtual` and *non-`virtual`* bases.[4] 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 the location of t_1 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

[4] We are not so interested in the textbook [14] 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.

virtual-bases. However, since a type is not always located at the same offset, it is necessary to apply a process called `this` -adjustment [13] in order to access a field introduced in a supertype. For example, a method of t_2 cannot be invoked on an object of type t_3, without first correcting the pointer to the object, coercing it to type t_2.

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

In hierarchies with virtual bases, even the weak conformance requirement cannot be satisfied together with object contiguity. In these cases, C++ uses *virtual base pointers* (VBPTRs) to tie memory segments of the same object. Gil and Sweeney [5] 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. 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 does have a VPTR.

Gil and Sweeney [5] proposed several optimizations of the standard C++ layout, which were then empirically evaluated by Eckel and Gil [4], 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 *bidirectional object layout* optimization reduces dynamic memory overhead but does not contribute to field access efficiency.

For the purpose of illustration, Figure 2 depicts a type hierarchy and its aggressive-inline C++ layout. The same hierarchy will be used below in Section 5 for demonstrating the new two-dimensional bi-directional layout. A C++ programmer is allowed to denote some of the inheritance edges as `virtual`. In the figure, inheritance edges $\langle \mathsf{B}, \mathsf{A} \rangle$ and $\langle \mathsf{C}, \mathsf{A} \rangle$ are virtual so that F has a single A sub-object. The virtual edges that were *inlined* in the aggressive-inline layout are marked in bold, while the other non-inlined virtual edges are dashed. The cells with a dot in Figure 2b represent VPTRs (VBPTRs were not drawn since they can be stored either in a class or in all of its instances).

The new 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). The most

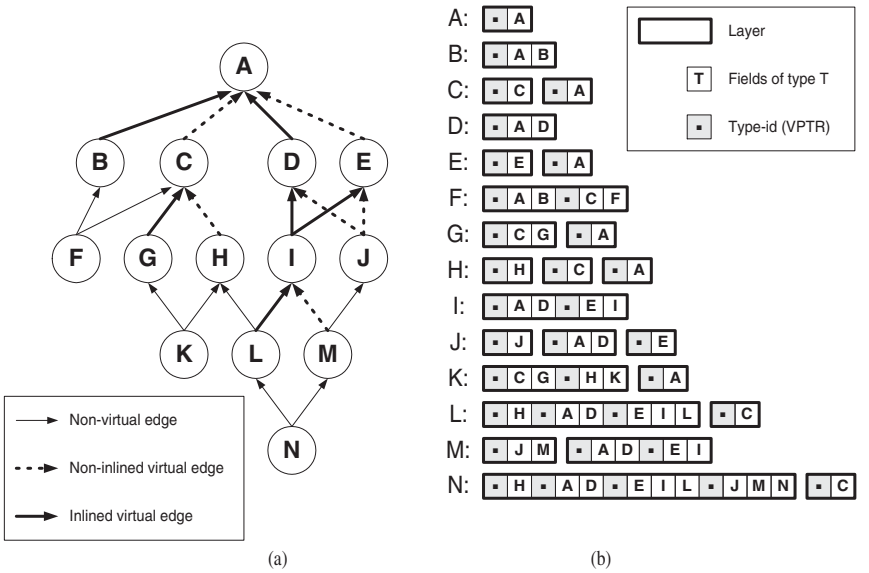

Fig. 2. A type hierarchy (a) with its aggressive-inline C++ layout (b)

interesting criterion for comparison with C++ and field dispatching is therefore field access efficiency. We shall see that the 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 7). 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 Two-Dimensional Bi-directional Object Layout

In the two-dimensional bi-directional object layout strategy each field defined in the type hierarchy has a two-dimensional address $\langle \ell, \Delta \rangle$. Coordinate ℓ, $1 \leq \ell \leq L$, is the field's *layer*, where L is the number of layers used by the type hierarchy. (The assignment of types into layers is the subject of Section 5.) Coordinate Δ is an integral offset of the field in its layer. We say that the layout is bidirectional since this offset may be either positive or negative.

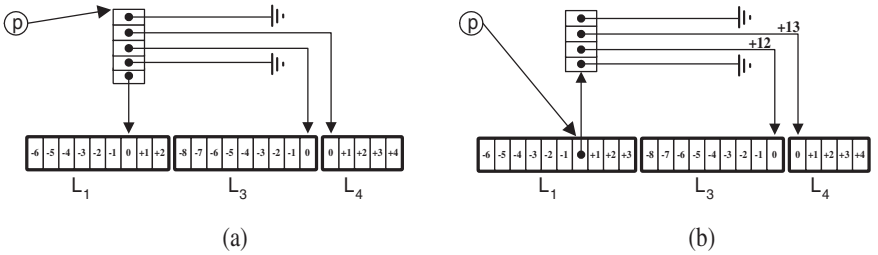

Fig. 3. The canonical (a) and the compact (b) two-dimensional bi-directional layout of an object from a 5-layer hierarchy. Layers L_2 and L_5 are empty in the depicted object

All fields introduced in the same type t are laid out consecutively: Their layer is the same as ℓ_t, the type's layer, while their offset is fixed with respect to Δ_t, the offset of the type. This section describes the actual object layout, which has three versions: the simple and not so efficient *canonical* layout, which is included for 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.

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

The canonical layout is demonstrated in Figure 3(a) for the case $L = 5$. 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.

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. More precisely, if an object has a field at a certain layer in offset $\Delta > 0$, then it also has fields in all offsets $0, \ldots, \Delta - 1$. 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.

A compiler algorithm for producing the runtime access code in the canonical layout is presented in 5canonical. Take note that the type t, the layer ℓ_t, and the offsets Δ_t and $\Delta_{\mathtt{f}}$ are computed at compile time. A *single* memory dereference is required to compute the field *address*.

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

Algorithm 1 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 ℓ_t be the unique layer of t // ℓ_t *is a positive integer*
3: Let integer Δ_t be the offset of t
4: Let $\Delta_{\texttt{f}}$ be the offset of `f` within its type // $\Delta_{\texttt{f}}$ *is a non-negative integer*
5: **Output**

```
int *layer_ptr = ((int **)p)[ℓ_t − 1];
int &r = layer_ptr[Δ_t + Δ_f];
```

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 3b gives an example of the compact layout of the same object of Figure 3a. 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 2 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 ℓ_t be the unique layer of t // ℓ_t *is a positive integer*
3: Let integer Δ_t be the offset of t
4: Let $\Delta_{\texttt{f}}$ be the offset of `f` within its type // $\Delta_{\texttt{f}}$ *is a non-negative integer*
5: **If** $\ell_t = 1$ **then**
6: **Output**

```
int &r = ((int *)p)[Δ_t + Δ_f];
```

7: **else**
8: **Output**

```
int *p1 = *((int **)p);
int layer_offset = p1[ℓ_t − 2];
int &r = p[layer_offset + Δ_t + Δ_f];
```

Algorithm 2 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 dererenced 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.

The LDT in the example of Figure 3 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 words. 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.

5 Computing Type Addresses

This section is dedicated to the algorithm for assigning field addresses. The main constraint to maintain is that all layers are contiguous in all types. It is always possible to find such an assignment, since each field can be allocated its own layer (as done in field dispatching).

Our objective is an assignment which minimizes L, the number of layers. One reason for doing so, is that the memory required for LDTs is $L \times n$. LDTs are source for static memory overhead in the compact layout, and dynamic memory overhead in the inlined layout.

However, our most important motivation is reducing the *likelihood of LDT fetches*, or in other words, inefficiency of field access. If the number of layers is one, then all fields can be retrieved without any dereferences. We note that if the number of layers is small, then an optimizing compiler might be able to pre-fetch and reuse layer addresses to accelerate field access.

Note first that each layer has a positive and a negative *semi-layer*, and that these semi-layers are independent for the purpose of allocation. To understand the constraints of allocation better, consider Figure 4a which gives the object layout for our running example.

We see in the figure that the hierarchy uses a total of two layers and three semi-layers. The first layer has at offset 0 the object type identifier and a positive and negative semi-layers. The second layer uses only the positive semi-layer. The arrows in the figure indicate the place where the semi-layer may continue.

Figure 4b shows the allocation of types to semi-layers which generates this layout: Seven types A, C, F, H, K, L, and N are in semi-layer 1 (positive side of the first layer). Semi-layer 2 (negative side of the first layer) includes five types: B, E, G, J, and M. Only D and I are in semi-layer 3 (positive side of the second

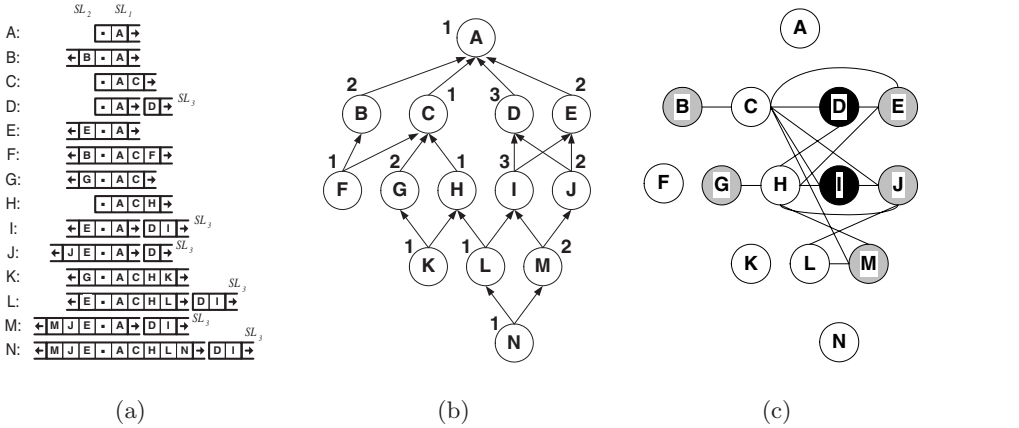

Fig. 4. The two-dimensional bi-directional object layout of the running example (a), the allocation of types in it to semi-layers (b), and the conflict graph with its coloring (c)

layer). The layout of type N for example, makes use of all three semi-layers, while the layout of D uses just semi-layers 1 and 3.

Notice the following points: **(i)** Semi-layers 1 and 2 comprising the first layer are in a fixed offset. Semi-layer 3 occurs at different offsets in different types. **(ii)** Each type is always placed in the same location in its layer. For example, E is located in the first location in semi-layer 2 in the layouts of all of its descendants: E, I, J, L, M, and N. **(iii)** The same location in the same semi-layer can be used for different types. For example, the first location of semi-layer 1 stores also the fields of B in the layout of F, and the fields of G in the layout of K. **(iv)** 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 semi-layer 1 in the layout of N and that $\mathsf{A} \succeq \mathsf{C} \succeq \mathsf{H} \succeq \mathsf{L} \succeq \mathsf{N}$.

The general question is whether two arbitrary types $a, b \in \mathcal{T}$ can be allocated to the same semi-layer, and what should their relative ordering in that semi-layer should be. Suppose first, without loss of generality, that $a \preceq b$. Then, whenever a appears, 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 location 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.

Figure 4c shows the *conflict graph* of our running example, where two types are connected by an edge if they are incomparable, yet have a common descendant. We see in the figure that no edges are incident on A. This is because A

is the root, and as such is comparable with all types in the hierarchy. Also, no edges are incident on the leaves F, K and N. The edge between C and E , for example, is due to their common descendant L.

A node coloring of the conflict graph provides a legal allocation. We of course seek a minimal coloring of this graph. Figure 4c gives a coloring of the conflict graph of the running example. A total of three colors are used: White nodes are allocated to semi-layer 1, grey to semi-layer 2, and black to semi-layer 3.

Algorithm 3 shows the general procedure for address allocation. Using a graph coloring heuristic, the algorithm computes the number of layers for the layout. Also, for each type t in the input hierarchy the algorithm returns ℓ_t, its layer and Δ_t, the base offset in the layer at which its fields are allocated. If $\Delta_t \geq 0$, then fields are allocated in ascending addresses. Otherwise, t is in the negative semi-layer, and field are placed in the addresses below Δ_t.

Algorithm 3 Produce the compact two-dimensional bi-directional layout of a hierarchy

Given a hierarchy $\mathcal{T}$ and $\preceq$, return the number of layers L, and compute ℓ_t and Δ_t for each type $t \in \mathcal{T}$

```
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: Let G ← ⟨T, E⟩ // G is the graph of conflicts between types
8: Let T ↦ [1, ..., s] be a coloring of the nodes of G

9: For all t ∈ T do // Compute the offset and the semi-layer of t
10:   Δt ← 0 // Compute the total size of proper ancestors in the same semi-layer as t
11:   For all p ∈ ancestors(t), p ≠ t do
12:      If p) = t) then // Ancestor p is in the same semi-layer as t
13:         Δt ← Δt + |p|
14:   ℓt ← ⌈t)/2⌉ // Layer l hosts colors 2l − 1 and 2l
15:   If t) mod 2 = 0 then // Even colored objects are laid out in negative semi-layers
16:      Δt ← −Δt − 1 // Offsets of negative semi-layers start at −1
17:   else if t) = 1 then
18:      Δt ← Δt + 1 // Offset 0 in layer 1 is reserved for the type-identifier

19: Return ⌈s/2⌉
```

Lines 3–3 compute the edges in the conflict graph. In the main loop, we consider the ancestors of each candidate. There is a conflict between any two of its ancestors if they are incomparable. The runtime of the inner loop should

(empirically) be close to linear, since the average number of ancestors in our hierarchies is small.

Next (lines 3–3) we compute the conflict graph and a coloring of it. We use a simple, greedy heuristic for finding this coloring. A favorable property of this heuristic is that the color groups tend to come out in descending order, i.e., $|-1(i)| \geq |-1(i+1)|$ for $i = 1, \ldots, s-1$. Since fields in the first layer can be accessed in a single indirection, the first layer should be as large as possible.

The next command block computes the layer of each type t, and its (positive or negative) offset within this layer. Lines 3–3 compute the total size of types which precede t in its semi-layer. After computing the layer number (line 3) 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 3–3). However, layer 1 is special since it contains the type identifier at offset 0 (lines 3–3).

6 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[5] 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 [17, 18]. (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.) To these we added Flavors, a 67-type hierarchy representing the *multi-inheritance core* of the Flavors language [8] benchmark used by Pugh and Weddell [9, 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 [4]

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 in average to each 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 1 below gives (among other information), the number of types in each hierarchy, and the maximal and average number of ancestors.

[5] 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.

Table 1. Statistics on the input hierarchies, including the number of colors and layers found by 5TD compared with the maximal anti-chain lower bound

Hierarchy $\langle \mathcal{T}, \preceq \rangle$	$n = \lvert\mathcal{T}\rvert$	ω [6]	s [7]	$\lceil \omega/2 \rceil$	$\lceil s/2 \rceil$	$\max(\theta_t)$ [8]	$\text{avg}(L_t)$ [9]	$\text{avg}(\theta_t)$ [10]
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

7 Experimental Results

This section presents the results of running 5TD on our data set. Since this algorithm depends on a graph-coloring heuristic (Line 3), we would like first to be assured by the output quality. 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 [15].

Table 1 compares the number of colors and layers with the predictions of the lower bound thus found.

Let $\omega_t = \max\{|P_t| \mid P_t \subseteq \text{ancestors}(t)$ 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 L. We see in the table that $s > \omega$ only in seven hierarchies: Flavors, Ed, LOV, MI: Orbacus Test, MI: HotJava, Geode and MI: Corba. In these seven cases, $s = \omega + 1$, 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 $\alpha = \max(\theta_t)$. As expected, 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 method dispatching techniques, such as selector coloring [3, 11]. 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 $n \times L$ is superior to that of the field dispatch matrix compressed using selector coloring.

The next two columns of Table 1 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 layers. This shows that the **(i)** Algorithm 3 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.

The theoretical complexity of Algorithm 3 is $O(n^3)$, since lines 3–3 may iterate in certain hierarchies over a fixed fraction of all possible type triplets. The runtime of the simple greedy graph-coloring heuristic is $O(n^2)$. In practice however, the algorithm runs much faster. By applying some rather straightforward algorithmic optimizations, e.g., considering in line 3 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 3 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 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.

The most important criterion for evaluating a layout scheme is 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. We can however derive other metrics to compare the costs of the new layout technique with that of prior art.

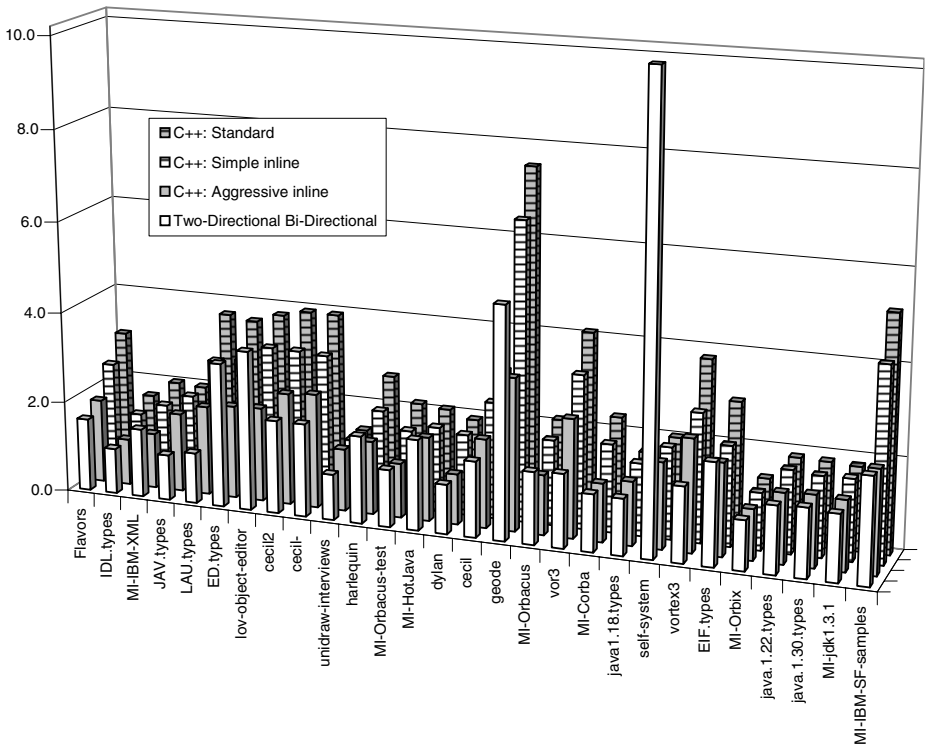

Fig. 5. Average no. of layers in different hierarchies

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 5 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. In making the comparison we bear in mind that the new scheme is both language-independent and space-optimal—properties which the C++ schemes do not enjoy.

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 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.

Table 2 shows the extra dynamic memory consumed by the various C++ layout schemes, specifically for VPTRs.

Table 2. No. of VPTRs using standard C++ layout, simple inline (S-Inline), and aggressive inline (A-Inline)

Hierarchy	Average			Median			Maximum		
	C++	S-Inline	A-Inline	C++	S-Inline	A-Inline	C++	S-Inline	A-Inline
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

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.

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. Since much empirical data is missing from our ensemble of hierarchies, we were inclined to make two major 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 L_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 $X_t(i)$, $i = 2, \ldots, L_t$ 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

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

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

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

Averaging over an entire type hierarchy, we define

$$A(k) = \frac{1}{n} \sum_{t \in \mathcal{T}} A_t(k) \tag{2}$$

Figure 6 gives a plot of $A(k)$ vs. k in four sample hierarchies in the layout schemes field dispatching, standard C++ layout, simple inline (S-Inline), aggressive inline (A-Inline), and two-dimensional bi-directional (TDBD). Values of $A(k)$ were computed using (1) and (2) 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

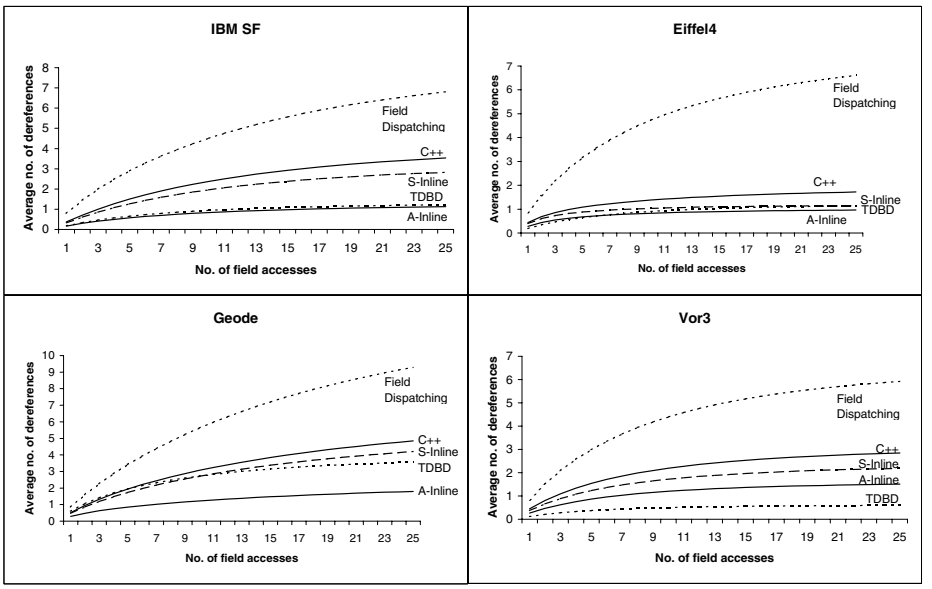

Fig. 6. Average no. of dereferences vs. no. of field accesses in four hierarchies

field dispatching. (Note however that the model does not take into account any optimizations which runtime systems may apply to field dispatching.)

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.

8 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 to our analytical model: the new scheme always improves upon

the field dispatching scheme and on the standard C++ layout model. Even compared to the highly optimized C++ layout, after performing aggressive inline, the new scheme still compares favorably.

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

The one-dimensional bi-directional layout of Pugh and Weddell's [9] 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

[1] T. Cargill, B. Cox, W. Cook, M. Loomis, and A. Snyder. Is multiple inheritance essential to OOP? Panel discussion at the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95) (Washington, DC), Oct. 1993. 329

[2] C. Chambers. The Cecil language, specification and rationale. Technical Report TR-93-03-05, University of Washington, Seattle, 1993. 334

[3] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–214, New Orleans, Louisiana, Oct. 1-6 1989. OOPSLA'89, ACM SIGPLAN Notices 24(10) Oct. 1989. 344

[4] N. Eckel and J. Y. Gil. Empirical study of object-layout strategies and optimization techniques. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 394–421, Sophia Antipolis and Cannes, France, June 12–16 2000. ECOOP 2000, Springer Verlag. 329, 330, 335, 342

[5] J. Y. Gil and P. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 256–275, Denver, Colorado, Nov.1–5 1999. OOPSLA'99, ACM SIGPLAN Notices 34(10) Nov. 1999. 329, 331, 335

[6] S. B. Lippman. *Inside The C++ Object Model.* Addison-Wesley, 2nd edition, 1996. 334

[7] B. Magnussun, B. Meyer, and et al. Who needs need multiple inheritance. Panel discussion at the European conference on Technology of Object Oriented Programming (TOOLS Europe'94), Mar. 1994. 329

[8] D. A. Moon. Object-oriented programming with flavors. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8, Portland, Oregon, USA, Sept. 29 - Oct. 2 1986. OOPSLA'86, ACM SIGPLAN Notices 21(11) Nov. 1986. 342

[9] W. Pugh and G. Weddell. Two-directional record layout for multiple inheritance. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 85–91, White Plains, New York, June 1990. ACM SIGPLAN, ACM Press. SIGPLAN Notices 25(6). 329, 331, 334, 342, 349

[10] W. Pugh and G. Weddell. On object layout for multiple inheritance. Technical Report CS-93-22, University of Waterloo—Department of Computer Science, May 1993. 331

[11] A. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 110–126, Vancouver, British Columbia, Canada, Oct.18-22 1992. OOPSLA'92, ACM SIGPLAN Notices 27(10) Oct. 1992. 344

[12] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997. 334

[13] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, Mar. 1994. 335

[14] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997. 334

[15] W. T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. The Johns Hopkins University Press, 1992. 343

[16] O. Zendra, C. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 125–141, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, ACM SIGPLAN Notices 32(10) Oct. 1997. 334

[17] Y. Zibin and J. Y. Gil. Efficient subtyping tests with PQ-encoding. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 96–107, Tampa Bay, Florida, Oct. 14–18 2001. OOPSLA'01, ACM SIGPLAN Notices 36(10) Oct. 2001. 329, 332, 342

[18] Y. Zibin and J. Y. Gil. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–160, Seattle, Washington, Nov. 4–8 2002. OOPSLA'02, ACM SIGPLAN Notices 37(10) Nov. 2002. 329, 332, 334, 342

[19] Y. Zibin and J. Y. Gil. Incremental algorithms for dispatching in dynamically typed languages. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'03)*, pages 126–138. ACM Press, 2003. 329, 332