Empirical Study of Object-Layout Strategies and Optimization Techniques

Natalie Eckel* and Joseph (Yossi) Gil**

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
natalie | yogi@cs.Technion.AC.IL

Abstract. Although there is a large body of research on the time overhead of object oriented programs, there is little work on memory overhead. This paper takes an empirical approach to the study of this overhead, which turns out to be significant in the presence of multiple inheritance. We study the performance, in terms of overhead to object size of three compilation strategies: separate compilation, whole program analysis, and user annotations as done in C++. A variant to each such strategy is the inclusion of pointers to indirect virtual bases in objects. Using a database of several large multiple inheritance hierarchies, spanning 7000 classes, several application domains and different programming languages we find that in all strategies there are certain classes which give rise a large number of compiler generated fields in their object layout. We then study the efficacy of the recently introduced inlining and bidirectional object layout optimization techniques, and show that an average saving of close to 50% in this overhead can be achieved.

1 Introduction

One of the most difficult challenges in the implementation of object oriented programming languages is to efficiently realize together multiple inheritance (MI) and dynamic dispatch. Indeed, it was believed [8] that it was impossible to efficiently introduce MI into C++, until proved otherwise [26].

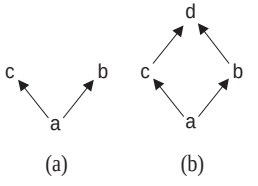


Fig. 1: Multiple inheritance and virtual inheritance.

The main difficulty is due to the fact that in presence of MI, an object of class a, can also serve as an instance of classes b and c, where no inheritance relationship exists between b and c, as in Fig. 1(a). The problem is complicated further in case b and c have a common ancestor, d, as depicted in Fig. 1(b). In this case d is called a *virtual base*, (of b, c, and a), and the inheritance links between b and d and c and d are called virtual inheritance[1] (VI).

To support this highly polymorphic nature of objects in presence of MI, the compiler is required to generate extra data fields in objects. Benchmarking shows that this overhead is significant [28]; and it can even double the memory footprint of some applications.

* Contact author

** Work done in part during a visit to the IBM T.J. Watson Research Center

[1] also *shared inheritance*

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 394–421, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In a recent theoretical paper, Gil and Sweeney [11] described the kinds of compiler generated fields: VPTRs —pointers to virtual functions tables (VTBLs), and VBPTRs— pointers to virtual bases. There are two kinds of VBPTRs. *Essential* VBPTRs , or for short, *e*-VBPTRs, point to *direct* virtual bases. A time-space tradeoff is offered by *inessential* VBPTRs , *i*-VBPTRs for short, which point to *indirect* virtual bases, i.e., virtual bases of virtual bases. *i*-VBPTRs make it possible to access an indirect virtual base in a single dereference operation, without going through any intermediate virtual bases. Optimization techniques for reducing the number of these fields were offered in the theoretical framework of [11].

Taking an empirical, yet language-independent, approach, our work continues the study of compiler generated fields. Compiler generated fields can be thought of as the *per-object* memory overhead of MI. It should be noted that MI incurs a bloat in two other kinds of memory overheads: *per-class* for storing class tables, and *per-hierarchy* for storing e.g., type inclusion information. Both are left outside the scope of this paper: Per-class overhead, which tends to be small, was thoroughly studied in the context of C++ in [28]; optimization techniques were suggested e.g., in [9]. See [15] for an empirical and theoretical study as well as a survey of research of the per-hierarchy overhead.

As described in Sec. 2, our study relies on a database of circa 7,000 classes. The main issues which interest us are summarized in the next few paragraphs. These also give forward references to the detailed discussion in the body of the paper.

Topology of large inheritance hierarchies. With such a large number of classes, it is becoming difficult to have an intuition on the structure of hierarchies. In Sec. 3 we define a suite of parameters of the topology of the directed acyclic graph (DAG) of inheritance. These parameters should help in gaining better understanding of the structure of inheritance hierarchies. By applying this suite to our database, we are able to reach some interesting findings on the topology of inheritance hierarchies, and explain some of the benchmarking results.

Separate compilation vs. whole program analysis. VI complicates MI not only by its presence, but also due to the uncertainty it generates. A compiler encountering class y inheriting from x, cannot be sure that x is not a virtual base of y before verifying in the whole inheritance hierarchy that there is no other class z that inherits *independently* from *both* x and y. In a separate compilation environment, the compiler is therefore inclined to make a worst case assumption and treat *all* inheritance links as virtual.

This research is the first to investigate the cost of this assumption which is a direct consequence of the separate compilation model. We will see in Sec. 4.4 that separate compilation more than doubles the number of compiler generated fields of a median object, while for some objects it incurs more than 40-fold increase in the number of the compiler generated fields!

In C++, the responsibility of distinguishing between virtual and non-virtual links lies with the programmer. However, this language design decision, which is probably as questionable as many other features of the language [24], does not make the situation much better. As explained in [11], with the absence of omniscient capacities, the programmer will tend to conservatively choose virtual over non-virtual inheritance. For example, in the almost standard IDL hierarchy (used e.g., in [10, 4, 28]), out of the 65

inheritance links (connecting 66 classes), 50 links were marked as virtual. Evidently, a whole program analysis reveals that only two inheritance links in this hierarchy are truly virtual!

Inessential VBPTRs. As explained in [11], all VI edges must be represented in the *traditional object layout scheme* using an *e*-VBPTR. However, to avoid an excessive number of dereference operations, which are known to be expensive in modern RISC processors, many compilers place *inessential* VBPTRs in object layouts. In Sec. 4.3 we give estimates on the actual memory overhead of this compiler implementation decision. We also comment on the expected number of dereference operation required to reach a virtual base in the absence of *i*-VBPTR.

Optimization techniques. Our findings show that even in the frugal whole program analysis strategy, there is a significant fraction of mammoth classes—classes with around a hundred compiler generated fields in each of their instances. This number increases to circa 250 if *i*-VBPTRs are included!

It was shown in [11] using a theoretical analysis and a number of small case studies that a significant reduction in these numbers might be possible, using a number of optimization techniques.

One of our primary objectives is to understand these optimization techniques better and to apply them to show that the memory overhead induced by the time-efficient *i*-VBPTRs can be reduced to a minimum. To this end, we give an empirical study of the efficacy of the following object layout optimization techniques.

ETRANS *Elimination of transitive virtual inheritance edges.* This technique is used mainly as a pre-processing stage for other, more advanced techniques.

DEVIRT *Devirtualizing single virtual inheritance edges.* Single virtual inheritance edges are edges designated as virtual by the programmer, but not serving as such.

S-INLN *Simple inlining of virtual bases. Inlining* means that no VBPTR is used to represent a virtual base in its descendant. Instead, the corresponding subobject is made part of the including object. The inlining is *simple* if no attempt is made to inline a virtual base into more than one of its descendants.
Note that devirtualization can also be thought of as inlining.

A-INLN *Aggressive inlining of virtual bases.* In aggressive inlining, a *maximal independent set* algorithm is used to maximize the number of descendants into which a virtual base is inlined. Since the best such algorithms are still exponential, care should be taken before their application. We therefore dedicate some attention in Sec. 6.2 to estimate the resources required for applying this advanced technique.

PAIRUP *Marriage of immediate non-virtual bases.* The bidirectional object layout suggested in [11] makes it possible to "marry" together a positive and an negative immediate, non-virtual bases of a class, whereby saving one VPTR in their representation. The algorithm for doing so is described in procedure *Pair-Up* of [11].

EPHEM *Ephemeral marriage of virtual bases.* Yet another optimization technique which uses the bidirectional object layout is the ephemeral marriage of virtual bases. The marriage is called *ephemeral* since virtual bases married in one class are not necessarily married in any of it descendants.

BIDIR *Combination of* PAIRUP *and* EPHEM.

H-PAIRUP *Marriage of immediate, non-virtual, hermaphrodite bases.*
This paper also proposes and investigates yet another new technique for the optimization of object layout. The *hermaphroditing* idea is similar to bidirectional object layout, except that directed classes, may have instances of both positive and negative directionality. Hermaphroditing increases further the marriage opportunities, but also requires means to distinguish at runtime between negatively and positively oriented objects. For example, this distinction could be made by using the sign bit of object pointers, so that negative pointers would be used to designate objects laid out in the negative direction. Modifications to the dispatch mechanism would be required to make the distinction between negatively and positively oriented VTBLs. H-PAIRUP refers to the hermaphrodite version of PAIRUP.

H-EPHEM *Ephemeral marriage of hermaphrodite virtual bases.*
This is the hermaphrodite version of EPHEM.

H-BIDIR *Combination of* H-PAIRUP *and* H-EPHEM.
In other words, this is the hermaphrodite version of BIDIR.

Table 1 gives the summary of the requirements and the potential savings of all of these techniques.

Technique	Requirements		Savings	
	whole program analysis	architecture support	VPTRS	VBPTRS
ETRANS	-	-	+	+
DEVIRT	+	-	+	+
S-INLN	+	-	+	+
A-INLN	+	-	+	+
PAIRUP	-	-	+	-
EPHEM	-	-	+	-
BIDIR	-	-	+	-
H-PAIRUP	-	+	+	+
H-EPHEM	-	+	+	+
H-BIDIR	-	+	+	+

Table 1: Techniques for optimizing object layout.

We see that inlining of all variants: devirtualization, simple and aggressive, requires whole program analysis. The primary kind of savings of savings is in VBPTRS. However, each saving in a VBPTR bears a potential for VPTR saving due to sharing. The efficacy of the inlining family of optimization techniques is discussed in Sec. 6.

On the other hand, bidirectional object layout techniques, whose benchmarking is described in Sec. 7, do not require whole program analysis, and can only save VPTRS. It is a rather surprising conclusion of our study that the savings of bidirectional techniques are in the same order of magnitude as DEVIRT and as A-INLN. Hermaphroditing is a new optimization technique described in detail in the sequel. The reader is referred to [11] for a detailed description of the other techniques. The efficacy of all the optimization techniques is summarized in Sec. 8. Finally, Sec. 9 gives the conclusions and outlines directions for further research.

2 Experimental Setting

This section discusses the database used in our benchmarking. Prior to this description, few words are in place regarding the special problems in making measurements of the

sort we are interested in. The main difficulty in benchmarking memory overhead due to MI is that of scale. Although measurements of time made on small and medium sized benchmark programs such as SPEC [12] and SPECjvm98 [25] remain meaningful when extrapolated to large application, no similar extrapolation can be made for memory usage. Small programs tend to make little use of MI. Consequently, their typical object layout is small and simple, leaving little or no room for optimization. Consider for example Driesen and Hölzle timing of virtual functions calls [10]. Their second largest application, groff, has only 92 classes, with only 48 inheritance relationships, with no MI, let alone VI. This data-point of 19,000 LOC of C++ is useless for our purposes, since in the absence of MI, all three kinds of memory overhead are minimal. In fact, the per-object memory overhead is reduced to one compiler generated field [11].

Obtaining a sample of large applications that make use of complex inheritance hierarchies, is difficult since such applications tend to be *propriety*, prohibitively *expensive*, and *unavailable* at all in source code form. Yet another bias in sampling is that as a result of the significant overhead of MI as encountered by every user, enforced by warnings from industry leaders (e.g., [5, 17, 18]), designers tend to restrain their needs for MI. Reducing this overhead by work such as this one is likely to increase the use of MI.

Our database selection started from the collection of large hierarchies which served Krall, Vitek and Horspool in benchmarking their algorithms for efficient type inclusions tests [16, 15].[2] This collection was used in other classical benchmarking papers, e.g., [9]. Of the eleven applications used in [15], four did not make any use of MI, and hence are not suitable for object-layout benchmarking (nor are they particularly appropriate for demonstrating type inclusion algorithms).

Also, a careful examination of the 225 nodes in the Java hierarchy used in [16, 15], shows that it wrongly assumes that Java interfaces inherit from class Object. We therefore disregarded this hierarchy, and replaced it with a hierarchy of around 1,700 classes of JDK 1.1. Since Java uses only a very restricted form of MI, we did not use this Java hierarchy at all in comparing the different strategies of MI. We however used it in our study of the topological structure of MI.

To these seven hierarchies, we added the class hierarchy of ISE version 4 [14] distribution of Eiffel [20]. Overall, the database consists of real word applications, covering a spectrum of usage of MI in various programming languages: C++ [27], Java [3], Eiffel [20], LOV (a language similar to Eiffel due to Verilog, France, a manufacturer of an OO CASE tool), Self [29] as well as Laure language of Caseau [6].

The hierarchies used are summarized in Table 2. The most important characteristics of each hierarchy are n, the number of classes (nodes), and m the number of inheritance relationships (edges). In total, the database has circa 8,500 nodes and 11,500 edges. Of these, around 7,000 classes and 9,500 edges were used in benchmarking MI.

Column r in the table gives the number of roots, i.e., classes which do not inherit from any other class. In single inheritance hierarchies r is simply the number of connected components, which will be 1 in the case the hierarchy is a tree. As can be seen from the table, r is strongly dependent in the programming language. Languages which allow multiple roots have a fair number of roots. However, overall only 2.9% of all classes are roots.

[2] Our thanks to Jan Vitek for providing access to this data.

The next column, α, the average number of parents of non-root classes, is indicative of the extent to which MI is used.

Hierarchy	Language	$n^{(i)}$	$m^{(ii)}$	$r^{(iii)}$	$\alpha^{(iv)}$
Unidraw	C++	613	476	147	1.02
Self	Self	1801	1838	51	1.05
Laure	Laure	295	315	1	1.07
JDK1.1	Java	1654	1927	89	1.23
Eiffel4	Eiffel	1999	2678	1	1.34
Ed	LOV	434	750	1	1.73
LOV	LOV	436	774	1	1.78
Geode	LOV	1318	2785	1	2.11
Total		8550	11543	293	1.39

(i) Number of classes
(ii) Number of inheritance edges
(iii) Number of root classes
(iv) Average number of parents of non-root classes

Table 2: Class hierarchies used in our experiments.

For all single inheritance hierarchies, $\alpha \equiv 1$. The hierarchies in Table 2 are sorted in ascending order of α. Thus, Laure, Self and Unidraw hierarchies are pretty close to a single inheritance hierarchy, while all the three applications written in LOV make an extensive use of MI.

It should be announced that some of the numbers presented in Table 2 *do not* agree with the corresponding values in Table 2 in [16] and Table 2 in [15]. The value of n for Unidraw, Self and Geode in Table 2 is greater by one than in [16, 15]. Also, the average number of parents (α) for Unidraw, Ed, LOV and Geode, does not agree with [16, 15], with differences ranging from 0.01 to 0.22. In view of these differences, we verified our software by having it rewritten by another person in another programming language; the results turned out to be the same in both implementations. This discrepancy could be explained by updates made by the authors to their database subsequent to publication, rather than a flaw in the experimental work of [16, 15]. However, it should be noted that the numbers in our table do agree with earlier reports on the same data (Table 1 of [9]).

Our data did not include information on the use of virtual vs. ordinary (repeated) inheritance in C++, nor on the equivalent of this distinction in the Eiffel family of languages. We were therefore inclined to assume that no repeated inheritance occured, i.e., no class is duplicated in any of its descendants. Thus, in the hierarchy of Fig. 1(b), we *always* use the semantics that there is only one occurrence of a subobject d in a, even though C++ gives way also to the semantics of two such occurrences. Our assumption agrees with the observation that repeated inheritance is a rarity, with the belief of some that "repeated inheritance is an abomination" [3], and the fact that many programming languages do not even support it, or as it is the case with Eiffel, require the programmer to go into extraordinary effort to make use of it.

No statistics on runtime memory usage was available. Indeed, for a library hierarchy such data would be meaningless, unless collected over a very broad array of applications which make use of this library. Moreover, measurement of memory usage in runtime opens the door to a debate between conflicting definitions, such as high water mark and total memory usage. Even further, many subtle issues such as fragmentation, tradeoffs between sizes of classes which tend to be instantiated together, etc., have to be dealt when dynamic runtime information is taken into account. Consequently, this research conten-

[3] words of an anonymous reviewer to [11]

ded itself with measurements based on static analysis; the problem of benchmarking runtime information is left for further research.

Our results are presented broken by overhead to object size and sometimes by hierarchy. The fundamental hypothesis is that since frequency of instantiation each class cannot be predicted, all the layout of *all* classes should be optimized. To obtain a single number summarizing a range of measurements it is sometimes convenient to make a

> UNIFORM INSTANTIATION HYPOTHESIS: *All classes are equally likely to be instantiated.*

In other words, with the absence of any other information, there is no reason to assume that one class is more likely to be instantiated than another class. Another hypothesis which can be used to compute such a single number is:

> LEAF INSTANTIATION HYPOTHESIS: *All leaf classes are equally likely to be instantiated, but internal classes are never instantiated.*

This hypothesis is supported by the observation that an abstract class is (usually) not a leaf, and on the proviso that library-user defined classes which are likely to be instantiated more frequently, tend also be leaves. As we will see, the differences in numbers between these two hypothesis is minute. This strengthens our belief that the static results we obtain carry their meaning to runtime performance.

3 The Topology of MI Hierarchies

Hierarchy	$\mu^{(i)}$	$\ell^{(ii)}$	$b^{(iii)}$	$d^{(iv)}$	$v^{(v)}$	$\nu^{(vi)}$
Unidraw	7.2%	78.5%	9	8	2	0.3%
Self	21.1%	63.0%	10	16	3	0.2%
Laure	3.5%	64.1%	8	11	11	3.7%
JDK1.1	19.3%	77.3%	10	8	19	1.1%
Eiffel4	23.4%	61.6%	10	14	63	3.2%
Ed	5.1%	50.7%	8	8	23	5.3%
LOV	5.1%	50.0%	8	9	24	5.5%
Geode	15.4%	55.5%	10	11	100	7.6%
Total	100%	64.1%	13	16	245	2.9%

(i) The relative weight of this hierarchy in the database $n/\sum n$
(ii) Percentage of leaf classes
(iii) Depth of the complete binary tree with n nodes
(iv) Depth of the hierarchy
(v) Number of virtual bases
(vi) Percentage of virtual bases

Table 3: Topological properties of the class hierarchies.

In designing algorithms and heuristics for MI hierarchies, it is important to understand the patterns of usage of MI in real applications.

Since such hierarchies are huge, it is difficult to gain any insight into their structure by a drawing their DAG which is often highly non-planar. Instead, we propose to study the structure of MI hierarchies by defining and computing topological properties of the inheritance DAG. Some such properties are summarized in Table 3.

Column μ gives the weight of this hierarchy in the database, computed based on the number of classes. Column ℓ gives the percentage of leaf classes. We already saw that there are very few root classes; now we see that the majority of all classes are leafs. This is a strong indication that hierarchies do not have a lattice like structure, despite the counter argument based on proper design considerations [13]. Perhaps this is also the

reason why Caseau [7] algorithm for type inclusion tests which tried to impose a lattice structure on the graph was inferior to other attempts at this challenge.

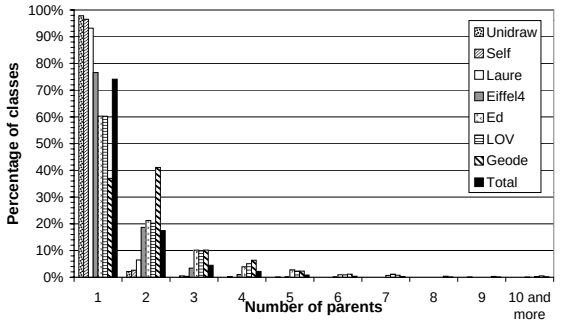


Fig. 2: Distribution of number of immediate base classes of non-root classes.

The depth of each hierarchy, i.e., the maximal distance in edges between a root and a leaf is denoted by d. By comparing d with the depth of a complete binary tree ($\lceil \lg(n+1) \rceil - 1$) we see that the hierarchies are very shallow. In fact, the depth never exceeds that of a balanced binary tree such as AVL.

The total number of virtual bases, i.e., classes which serve as an ancestor to some other class along more than one path, is given in column v of Table 3. Note that the number of virtual bases is relatively small. Overall, only about 3% of all classes are virtual bases. However, as we shall see later, the majority of classes have at least one virtual base. As a result, this small fraction of virtual classes has a huge impact the object size of many more classes.

Some more topological properties of our hierarchies are given in Table 4.

Hierarchy	$\alpha^{(i)}$	$\beta^{(ii)}$	$\gamma^{(iii)}$
Unidraw	$1.02^{\pm 0.15}$	$3.61^{\pm 5.18}$	$0.02^{\pm 0.18}$
Self	$1.05^{\pm 0.33}$	$2.76^{\pm 11.2}$	$0.73^{\pm 0.47}$
Laure	$1.07^{\pm 0.27}$	$2.97^{\pm 1.77}$	$2.86^{\pm 1.15}$
JDK1.1	$1.23^{\pm 0.57}$	$5.13^{\pm 22.9}$	$0.52^{\pm 0.74}$
Eiffel4	$1.34^{\pm 0.75}$	$3.49^{\pm 18.8}$	$2.49^{\pm 2.61}$
Ed	$1.73^{\pm 1.17}$	$3.50^{\pm 7.10}$	$3.79^{\pm 2.79}$
LOV	$1.78^{\pm 1.28}$	$3.55^{\pm 7.22}$	$3.99^{\pm 3.04}$
Geode	$2.11^{\pm 1.50}$	$4.76^{\pm 17.1}$	$8.37^{\pm 6.30}$
Total	$1.39^{\pm 0.92}$	$3.74^{\pm 15.6}$	$2.62^{\pm 4.02}$

(i) Average number and standard deviation of parents of non-root classes
(ii) Average number and standard deviation of children of non-leaf classes
(iii) Average number and standard deviation of virtual base classes

Table 4: Additional topological properties of the class hierarchies.

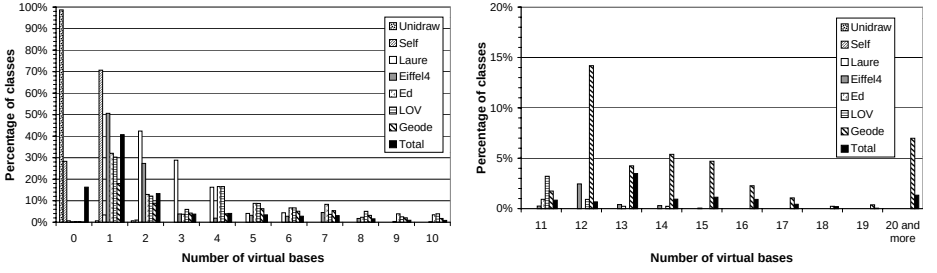
Column, α is similar to the column with the same title in Table 2, gives both the average and the standard deviation of the number of parents of non root classes. Note that both these values are not much greater than 1. By examining the distribution of the number of parents in Fig. 2, we see that more than 70% of all classes have only one parent, while for the Unidraw, Self and Laure hierarchies, this number increases to over 90%. Further, more than 90% of all classes have no more than two parents. Even for Geode, which has the least number of classes with one parent, close to 80% of all classes have at most two parents. We see that classes with large number of parents are rare, even in hierarchies which make quite an extensive use of MI.

We argue that the impact of MI on classes is gradual. Although classes tend to have a small number of immediate parents, these parents might also use MI, etc. This phenomena can be seen by examining the number of root-classes from which non-root classes inherit, directly or indirectly, which is $20.59^{\pm 4.49}$ in Geode, even though for

this hierarchy $\alpha = 1.05^{\pm 0.33}$. Interestingly, for this hierarchy, $\alpha^d = 1.05^{16} = 2.1827$, which is still much smaller than the average number of roots. In other words, the nodes in which multiple inheritance occurs are located in critical and influential junctions of the hierarchy.

Dual to the α parameter is β, the number of children of non-leaf classes. As can be seen from Table 4, the standard deviation of β is quite large, often greater than its average value.

Finally, the γ column of Table 4 gives the average number of virtual bases, direct and indirect that each class has. The distribution of the number is depicted in Fig. 3. In Unidraw there are very few classes with virtual bases. This is partly explained by the fact that as many as 24% of the classes of Unidraw are roots. With this exception, we see that even though virtual bases are scarce in general, many more classes have virtual bases. In the Self hierarchy for example, even though there are in total only 3 virtual bases, over 70% of all classes use one or more of these virtual bases.



(a) 1. . . 10 virtual bases (b) 11. . . 20 and more virtual bases

Fig. 3: Distribution of number of virtual bases of a class.

In single-root hierarchies, virtually all classes have at least one virtual base. Further, in Eiffel4, 27% of all classes have two virtual bases. There is also a significant fraction of classes with a very large number of virtual bases. In Geode over 40% of classes have 10 or more virtual bases. The large standard deviation in the number of virtual bases is also worth noting, since the overhead in object size may increase quadratically with the number of virtual bases [11].

4 Traditional Compilation Strategies

There are two major strategic decisions in the compilation of MI. First, it should be decided whether whole program analysis could be done, at the cost of slowing down the compilation process. We will use the notation WHO for a whole program analysis compilation strategy.

The alternative to WHO is to use a separate compilation strategy, which will be denoted as SEP. SEP which appears to be used by compilers such as ISE Eiffel [19, pp.339-340], is faster, but may produce less efficient executable. In particular, since a

SEP compiler has no way of predicting that an inheritance edge will not turn out to be virtual, it must conservatively assume that *all* inheritance edges are virtual.

The traditional C++ compilation model, denoted as CC, sits somewhere in between SEP and WHO. On the one hand, the programmer must annotate all virtual inheritance as such by using the virtual keyword. On the other hand, more advance optimization techniques are impossible since no whole program information is available until link time.

This section focuses on CC and SEP. As we shall see shortly, WHO is nothing but CC with the application of devirtualization. We postpone the discussion of WHO to Sec. 6.3.

The second decision that must be made is whether *i*-VBPTRs be included in object layout. The advantage is in maintaining the time efficient "single dereference distance" between an object and each of its subobjects. A "+" superscript will be used to denote a strategy variant which uses *i*-VBPTRs: SEP$^+$, WHO$^+$, and CC$^+$.

The alternative is to optimize memory by omitting those, but with time overhead of following a chain of pointers in an upcast to a non-direct virtual base. A "−" superscript on the name of strategy will be used to denote omission of *i*-VBPTRs.

4.1 Kinds of Inheritance Edges

To simulate the CC in our hierarchies which are mostly non-C++, it is necessary to *automatically* generate the programmer's virtual annotations.

In general, it is difficult to predict this kind of human decisions, which might be whimsical or influenced by a personal sense of style. For example, the IDL hierarchy unnecessarily used a large number of virtual inheritance, since it is designed also to be an elegant application framework.

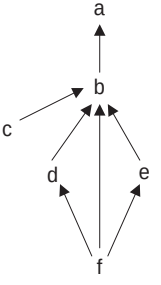


Fig. 4: The three kinds of inheritance edges.

In our to understand how programmer annotations are generated automatically consider the hierarchy graph of Fig. 4. It makes sense to assume that edge (b, a) will *not* be annotated as virtual. We will automatically denote edges of this sort as *non-virtual*. We have that (f, d) and (f, e) are non-virtual as well. With our assumption forbidding repeated inheritance, we have that (d, b), (e, b) and (f, b) are virtual. Edge (c, b) poses an interesting dilemma. A whole program analysis would reveal that this edge is non-virtual. However, a prudent programmer is likely to mark it as virtual in anticipation of code evolution. A class which served once as a virtual base is likely to serve as such again in the future. This is confirmed by our findings in Sec. 3 that virtual bases tend to serve multiple times as such. In this example, the introduction of a new class g inheriting from both c and d will *require* that (c, b) is marked as virtual. We will call (c, b) and other edges of its kind *potentially virtual*. In order to simulate the CC strategy we assume that potentially virtual edges are annotated virtual. In WHO, these edges will be considered non-virtual.

Table 5 gives the breakdown of inheritance edges into these three kinds in our hierarchies. With close to two-thirds of all edges, non-virtual inheritance dominates the two other categories.

Hierarchy	Non-virtual		Virtual		Potentially virtual	
	m_n [(i)]	$\frac{m_n}{m}$ [(ii)]	m_v [(iii)]	$\frac{m_v}{m}$ [(iv)]	m_{pv} [(v)]	$\frac{m_{pv}}{m}$ [(vi)]
Unidraw	471	98.9%	5	1.1%	0	0%
Self	1627	88.5%	6	0.3%	205	11.2%
Laure	268	85.1%	30	9.5%	17	5.4%
Eiffel4	1709	63.8%	464	17.3%	505	18.9%
Ed	423	56.4%	135	18.0%	192	25.6%
LOV	423	54.7%	140	18.1%	211	27.3%
Geode	1305	46.9%	763	27.4%	717	25.7%
Total	6226	64.7%	1543	16.0%	1847	19.2%

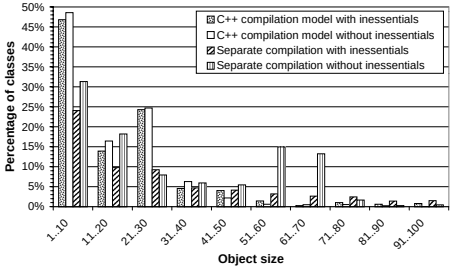
[(i)] Number of non-virtual edges
[(ii)] Fraction of non-virtual edges
[(iii)] Number of virtual edges
[(iv)] Fraction of virtual edges
[(v)] Number of potentially virtual edges
[(vi)] Fraction of virtual edges

Table 5: The kinds of inheritance edges.

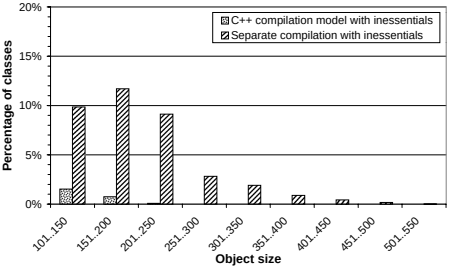
There are around 20% of potentially virtual edges, that will be devirtualized in WHO. The number of virtual inheritance, is the smallest and stands at 16%. These edges are the main target of our inlining optimization technique. It is also interesting to note that non-virtual inheritance decreases and virtual inheritances increases, as α increases. On the other hand, there seems to be no direct relationship between α and the fraction of potential-virtual edges.

4.2 Object Size in Different Compilation Strategies

For brevity, we will use the term *object size* to refer to the total number of compiler generated fields in the layout of objects of a certain class. No confusion will arise since our database measurements do not include information on the other kinds of fields stored in an object. Another slight abuse of terminology will be in the use of "*class size*" in the meaning "number of compiler generated fields in objects of the class".



(a) object size 1... 100



(b) object size 101... 550

Fig. 5: Distribution of object size in SEP$^+$, SEP$^-$, CC$^+$, and CC$^-$.

Fig. 5 show how object size values are distributed in the four main strategies: CC^+, CC^-, SEP^+, SEP^-. The X-axis gives the object size in terms of the number of compiler generated fields. The Y-axis is the percentage of classes out of the total of all classes in our database with the corresponding size.

In CC^+ and CC^-, the object size of most classes is under 40. In particular, CC^-, doesn't produce objects greater than 100 in size. Around 2.5% of all classes in CC^+ give an object size in the range of 100 to 250. In SEP, the percentage of mammoth objects begins to grow. Around 30% of objects have size between 50 and 70 in SEP^-. In SEP^+ around 35% of objects will have size between 100 and 550.

With the objective of showing that both time and memory efficient object layout is possible in C++, we choose CC^+ as the *baseline* for benchmarking optimization techniques. Henceforth, unless clearly specified otherwise, all results are presented in comparison to this strategy. Note that this choice of a baseline sets the bar at a pretty high level, since the size of 50% of all classes is 10 or less.

Fig. 6 examines the distribution of object size in CC^+ more carefully.

We see that there are three main peaks to the distribution– at object size 1–5, at around 20, and at around 50. In addition, a smaller and almost equal number of classes can be found at all object sizes in the range 20–150.

In total we see that there is a very significant portion of very small classes: 30% of all classes have at most three compiler generated fields. Moreover, the size of 85% of all classes have no more than 30 compiler generated fields.

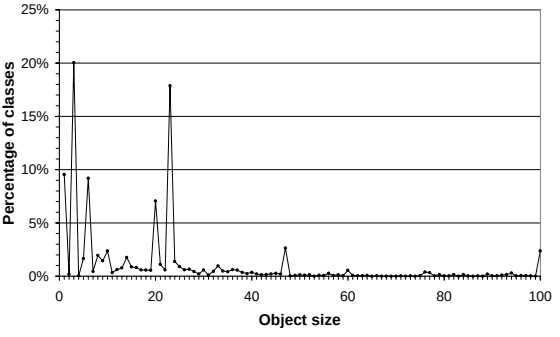


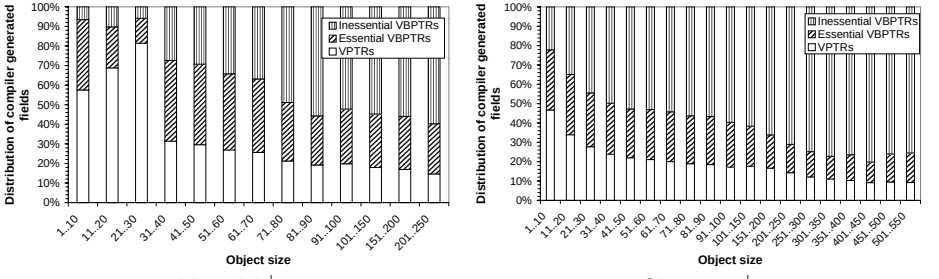
Fig. 6: Distribution of object size in CC^+.

4.3 Distribution of Compiler Generated Fields

As explained before, there are three kinds of compiler generated fields: VPTRs, *e*-VBPTRs, and *i*-VBPTRs. Fig. 7 shows how the total overhead is broken down into these three kinds for different object sizes.

Even though the graph in Fig. 7(a) shows the values for the CC^+ strategy, it is easy to infer from it the distribution in CC^-, which is the same as CC^+, except that no *i*-VBPTRs occur. It is important to note that in the bulk of the objects, i.e., objects of size no greater through 30, 50–80% of the overhead is due to VPTRs. This is indicative of the potential for savings by bidirectional layout techniques of classes of this size. To reduce the size of larger classes, inline techniques must be activated.

A similar breakdown for SEP^+ is shown in Fig. 7(b). Since in SEP^+ all edges are marked virtual, we have that for every VPTR in a subobject, there is at least one *e*-VBPTR in the object or in a containing subobject pointing to this subobject. In this strategy a subobject corresponding to a virtual base will even have two or more VBPTRs pointing



(a) in CC$^+$ (b) in SEP$^+$

Fig. 7: Distribution of compiler generated fields.

to it. The number of *i*-VBPTRs could be even greater. Therefore, the fact that the majority of the overhead in this strategy is due to VBPTRs is not surprising.

Fig. 8 sheds light from a different angle on the cost of including *i*-VBPTRs in SEP and in CC. This graph is *accumulative* in the sense that a data point (x, y) means that x percent of all classes experience an increase of y percent or more in size due to *i*-VBPTRs. More accumulative graphs will follow.

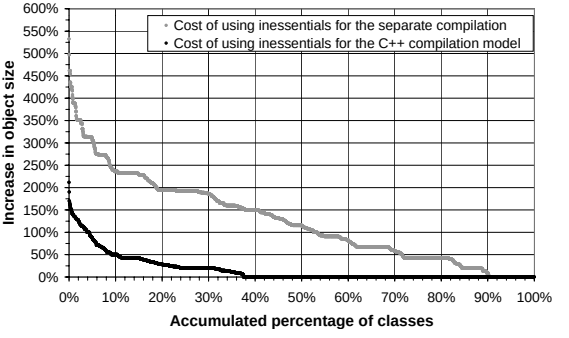


Fig. 8: Accumulative distribution of cost of using inessentials for CC and SEP.

In CC, we see that although there are some classes whose size is almost tripled by *i*-VBPTRs, over 60% of all classes do not have *i*-VBPTRs at all, while there are 20% of the classes whose size is increased due to *i*-VBPTRs by more than 30%. Thus, in the majority of classes, *i*-VBPTRs induce only a modest increase to object size. This gives a good justification to the decision of numerous compiler writers to use *i*-VBPTRs in C++ compilers. However, since applications may instantiate classes at varying frequencies, there should be a strong effort to optimize the use of *i*-VBPTRs in those 10% of classes in which the increase is 50% or more. In contrast, the use of *i*-VBPTRs in the SEP model may put a heavy strain on resources. The size of 50% of all classes is at least doubled in switching from SEP$^-$ to SEP$^+$.

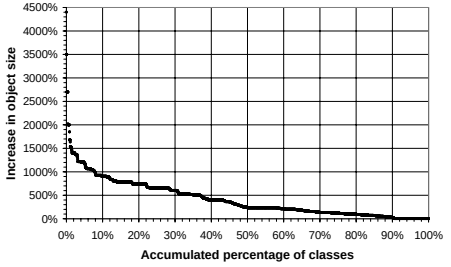
4.4 Cost of Separate Compilation

It is quite apparent from Fig. 5 that SEP generates much larger objects than CC. It is however interesting to compare the size of the same objects in these two different strategies.

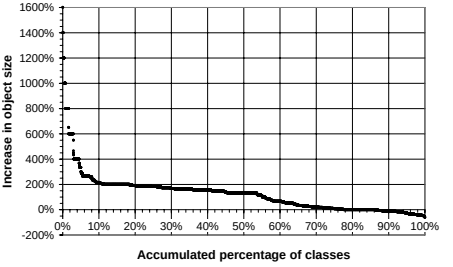
Fig. 9(a) shows the increase in object size when SEP$^+$ is chosen over CC$^+$. With close to 40% of classes suffering a six-fold increase, and some classes increasing by

a factor of over forty, the CC^+ strategy seems to be truly impractical. In a separate compilation model the decision to store *i*-VBPTRs should not be made lightly.

The alternative, namely the SEP^- strategy, incurs a cost of an increase in the time to access indirect virtual bases.



(a) choosing SEP^+ over CC^+



(b) choosing SEP^- over CC^+

Fig. 9: Accumulative distribution of increase in object size.

Let us now compare SEP^-, the more memory-efficient version of SEP, with CC^+, the memory-wasteful version of CC, as done in Fig. 9(b). Although there is a small fraction (less than 10%) of classes in which SEP^- is more efficient than CC^+, in the majority of classes CC^+ is much more efficient than SEP^-. Phrased differently, C++ without the virtual annotation by user, and without whole program analysis will incur a median increase in object size of over 100%!

5 Eliminating Transitive Inheritance

ETRANS is a preliminary step to more advanced optimization techniques, in which all transitive virtual inheritance edges are eliminated. Since by assumption there is no repeated inheritance in our hierarchies, this step is basically to remove *all* transitive edges. Beyond the obvious simplification of the hierarchy graph, the elimination of transitive virtual edges reduces object size. ETRANS can be also implemented in SEP, since even in separate compilation it is assumed that whenever a class is compiled, information on all of its ancestors is available.

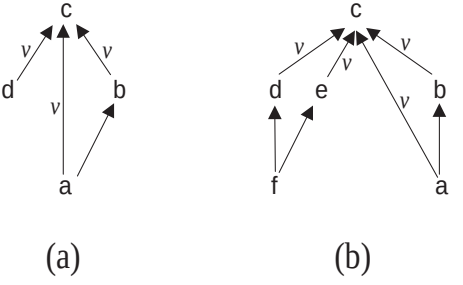


Fig. 10: Change in status of edges due to ETRANS.

We will use the name of a compilation strategies as a subscript in a name of an optimization technique to emphasize the environment where the technique is applied, for instance ETRANS_{CC^+} will notify ETRANS applied in CC^+ environment.

ETRANS reduces object size due to the fact that transitive edges, by definition, make diamonds hierarchies (Fig. 1(b)). Typically, an elimination of a transitive edge will also eliminate a diamond. This could make virtual edges into potentially virtual, or even non-virtual. In addition, potentially virtual edges may become non-virtual.

Consider for example the hierarchy of Fig. 10(a). By eliminating the transitive edge (a, c), the edge (b, c) changes its status from virtual to non-virtual.

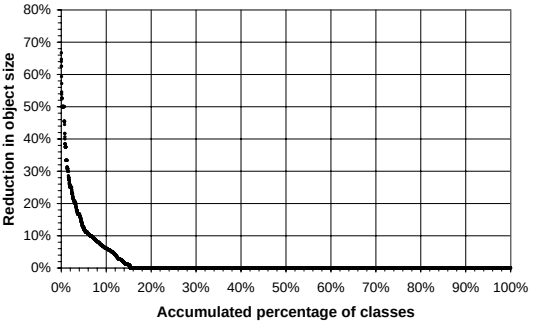


Fig. 11: Accumulative distribution of efficacy of ETRANS$_{CC+}$.

Also, the edge (d, c) changes its status from potentially virtual to non-virtual. Removing the transitive edge (a, c) in the hierarchy of Fig. 10(b), changes the status of the edge (b, c) from virtual to potentially virtual.

Table 6 describes the changes in distribution of the inheritance edges by kind due to ETRANS$_{CC+}$. The fact that as many as 4.1% of all edges were eliminated indicates that programmers do make use of transitive inheritance edges, probably as a matter of design style, since transitive inheritance contributes only very little to the semantics. It is evident however that there are hierarchies in which no transitive edges were used, although at least in C++ this is not forbidden by the programming language.

		Non-virtual		Virtual		Potentially virtual	
Hierarchy	Δm[(i)]	$\frac{m_n}{m}$[(ii)]	Δm_n[(iii)]	$\frac{m_v}{m}$[(iv)]	Δm_v[(v)]	$\frac{m_{pv}}{m}$[(vi)]	Δm_{pv}[(vii)]
Unidraw	0.0%	98.9%	0.0%	1.1%	0.0%	0.0%	0.0%
Self	0.0%	88.5%	0.0%	0.3%	0.0%	11.2%	0.0%
Laure	0.0%	85.1%	0.0%	9.5%	0.0%	5.4%	0.0%
Eiffel4	-1.6%	65.8%	+1.5%	15.6%	-11.4%	18.6%	-2.8%
Ed	-3.9%	70.3%	+19.9%	13.6%	-27.4%	16.1%	-39.6%
LOV	-3.5%	61.8%	+9.2%	14.3%	-23.6%	23.8%	-15.6%
Geode	-10.7%	60.7%	+15.6%	16.8%	-45.3%	22.6%	-21.8%
Total	-4.1%	71.3%	+5.6%	11.6%	-30.4%	17.0%	-15.1%

[(i)] Relative change in the total number of edges
[(ii)] New fraction of non-virtual edges
[(iii)] Relative change in the number of non-virtual edges
[(iv)] New fraction of virtual edges
[(v)] Relative change in the number of virtual edges
[(vi)] New fraction of potentially virtual edges
[(vii)] Relative change in the number of potentially virtual edges

Table 6: The impact of ETRANS$_{CC+}$ on the distribution of inheritance edges.

About one in three virtual edges was either eliminated or made into non-virtual or potentially virtual. This brought the percentage of virtual edges from 16% (Table 5) down to 11.6%. The reduction in the percentage of potentially virtual edges was more modest (17% instead of 19.2%). The most dramatic changes in the distribution were of course in Geode since this hierarchy had the largest fraction of transitive edges.

In order to describe the impact of an optimization technique on the object size we will use the term *efficacy*.

Definition 1. *The* efficacy *of optimization technique for a certain class is the relative reduction in object size of a class due to application of the technique.*

Fig. 11 shows the efficacy of ETRANS$_{CC^+}$. As can be seen in the figure, ETRANS is not a particularly effective optimization technique. Even though there is a very small fraction of classes which enjoy a significant reduction of 40% or more, overall only 8% of all classes experience a saving of 8% or more in their size.

6 Inlining Techniques

Consider the class hierarchy of Fig. 12. In this hierarchy, classes a and e are virtual bases and edges (b, a), (c, a), (d, a), (e, a) (h, e) and (i, e) are all virtual. Also, (f, a) is designated as potentially virtual in the CC strategy since it is incident on a virtual base. A whole program analysis will reveal that (f, a) is not virtual and will *devirtualize* it.

After DEVIRT was applied, S-INLN optimization, chooses exactly one of virtual edges incident on a virtual base, and inlines this base down this edge. In this example, a virtual base a may be inlined into any one of b, c, d or e by S-INLN. A-INLN improves on S-INLN by choosing a *maximal* subset of virtual edges incident on a virtual base, such that this virtual base can be inlined into all of them. A-INLN will inline a into b, e and f, which is the maximal set of children of a, such that no two of them share a descendant.

Note that devirtualization and the other inlining optimization techniques ([11]), are greatly simplified thanks to the absence of duplicate inheritance. Class a could not be inlined into e under no circumstances, if k also had a repeated inheritance link to e.

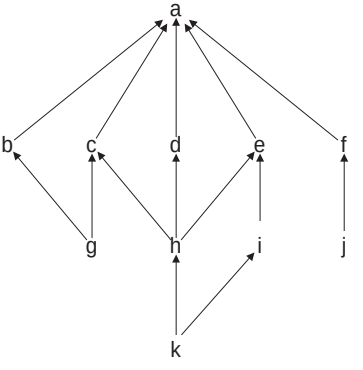


Fig. 12: A class hierarchy demonstrating different levels of inlining.

Every time inlining takes place, an *e*-VBPTR is eliminated. Inlining also has the potential of eliminating *i*-VBPTRS since it cuts by one the number of dereference operations of *e*-VBPTR required to reach a virtual base.

Since (e, a) and (i, e) are virtual edges, the *i*-VBPTR leading from i to a can be eliminated as soon as a is inlined into e. Inlining has also the potential to eliminate VPTRS. In the example, when a is inlined into f, the VPTR of a f object can be shared with the VPTR of its a subobject.

6.1 The Extent of Inlining

Table 7 gives the fraction of edges which are inlined by each of these three techniques.

Hierarchy	DEVIRT	SINLN$_{CC^+}$	AINLN$_{CC^+}$
Unidraw	0.0%	0.4%	0.4%
Self	11.2%	0.2%	0.2%
Laure	5.4%	3.5%	5.1%
Eiffel4	18.6%	3.5%	9.5%
Ed	16.1%	2.4%	6.0%
LOV	23.8%	2.8%	6.3%
Geode	22.6%	3.1%	8.6%
Total	17.0%	2.4%	6.3%

Table 7: The fraction of edges inlined by DEVIRT, SINLN$_{CC^+}$ and AINLN$_{CC^+}$.

Comparing this to Table 6 we see that all potentially virtual edges were inlined by DEVIRT. We also see that SINLN$_{CC^+}$ inlines only 2.4% of all edges, i.e., less than one in four of all virtual edges. There are significant differences in the extent of inlining of SINLN$_{CC^+}$. Two out of three virtual edges are inlined in Self, while less than one in five edges are inlined in Geode. This variance and the small extent of SINLN$_{CC^+}$ can be explained by the huge standard deviation in the number of children (column β in Table 4). Since S-INLN inlines into only one descendant, its impact would be smaller in all virtual bases with large number of descendants.

Note that the variance between the hierarchies in the extent of inlining by AINLN$_{CC^+}$ is much smaller than that of SINLN$_{CC^+}$. Overall, AINLN$_{CC^+}$ is successful in inlining more than one in two virtual edges.

6.2 Implementation of Aggressive Inlining

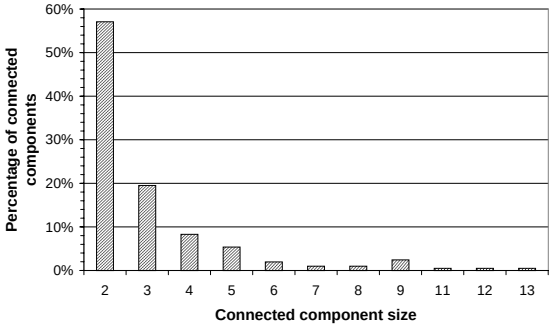


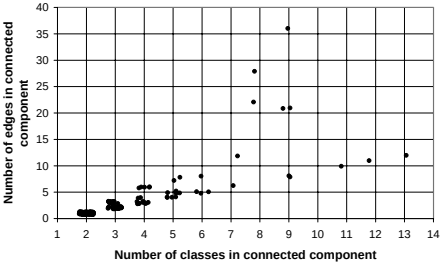
Fig. 13: Distribution of size of connected components.

As we said, A-INLN uses a maximal independent set algorithm. Two immediately derived classes of a virtual base are dependent if they share a descendant, because the virtual base must occur exactly once in that descendant. For each virtual base A-INLN creates a dependency graph of all of its immediately derived classes. This graph is then broken into its connected components, and the maximally independent set of nodes in each component is then found. We found that the vast majority (98%) of components are small, having no more than 13 nodes. For such small graphs, an exhaustive search algorithm with minor optimizations can be used to find the maximally independent set.

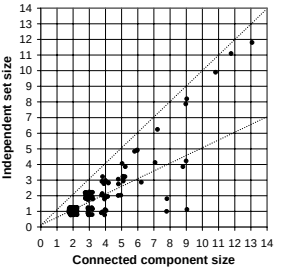
From Fig. 13 describing the distribution of size of small connected components, we see that close to 90% of all connected components have no more than 4 nodes—which makes the exhaustive search approach even more appealing.

Fig. 14(a) is a scatter diagram giving the number of edges in those components. Since the coordinates of all points are integral and drawn from a small range, many data

points may coincide at the same grid location. To visualize these clusterings, we add a small random *perturbation* $-0.25 \leq \epsilon \leq 0.25$ to all values. This technique is used in all subsequence scatter diagrams as well. We see that most of the components are sparse. In fact, 82% of all components are trees. Now, since a tree is also a bipartite graph, the maximally independent set has at least half of the nodes. Fig. 14(b) describes the size of the maximal independent sets found in small connected components. The two diagonal guidelines drawn on the grid show that in the maximal independent set is at least half in size of most connected components. In fact, there are many connected components in which all but one node take part in the maximal independent set.



(a) Distribution of edges and nodes



(b) Distribution of size of maximal independent sets

Fig. 14: Small connected components.

6.3 The Efficacy of Inlining Algorithms

Even though the extent of inlining is interesting, it is much more important to determine how much inlining reduces object size.

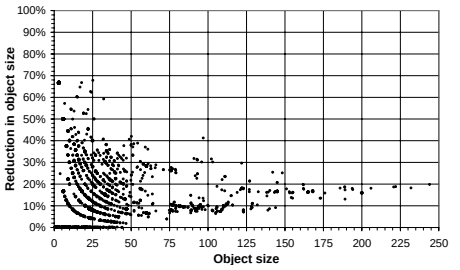
For the purpose of presentation in this section we will include DEVIRT in SINLN$_{CC^+}$ and in AINLN$_{CC^+}$. Thus, SINLN$_{CC^+}$ will be at least as good as DEVIRT and AINLN$_{CC^+}$ will be at least as good as SINLN$_{CC^+}$.

Fig. 15(a) is a scatter diagram showing the reduction in object size due to DEVIRT. It is important to note that DEVIRT reduces the size of *all* classes with 45 or more compiler generated fields. Also, even though there are small classes in which no savings are made, there is a cluster of large classes of size 50 or more with savings in the range of 10-20%.

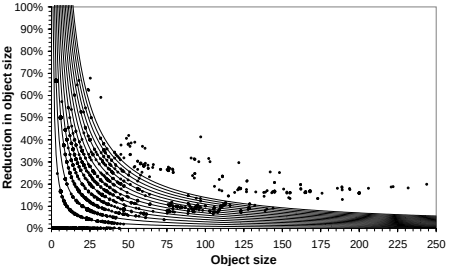
A somewhat closer look at Fig. 15(a) reveals that the points appear to lie on hyperbolic curves. This is by no means a coincidence!

A saving of f fields in a class of size b will place a point $(b, f/b)$ on the graph. In other words, all classes with a saving of f fields can be found on the hyperbola $y = f/x$. To illustrate this, we redraw Fig. 15(a) in Fig. 15(b) with the hyperbolic curves corresponding to $f = 1, \ldots, 14$.

Fig. 16(a) is a similar scatter diagram for SINLN$_{CC^+}$. This time we see that savings occur in all classes whose size is greater than 30. Savings in large classes with more than 50 compiler generated fields is clustered in the 25%-30% range.



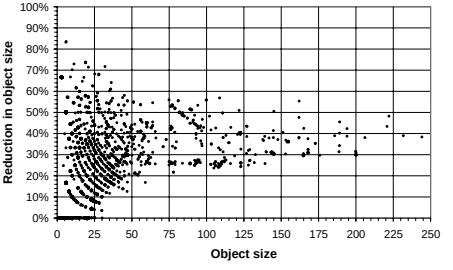
(a) Efficacy of DEVIRT



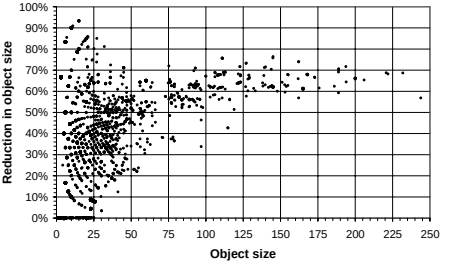
(b) Fig. 15(a) with the first 14 hyperbolic curves

Fig. 15: Eifficacy of DEVIRT.

As can be seen from Fig. 16(b) AINLN$_{CC+}$ guarantees savings at some level for all classes whose size is greater than 25. This time, the typical saving in large classes is around 60%, and even more than that for really large classes.



(a) SINLN$_{CC+}$



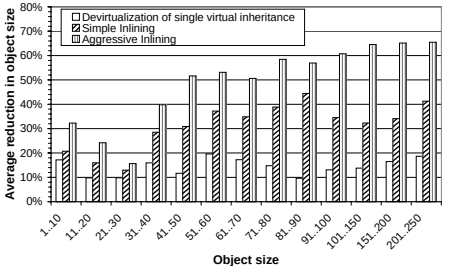
(b) AINLN$_{CC+}$

Fig. 16: Efficacy of inlining techniques in CC$^+$.

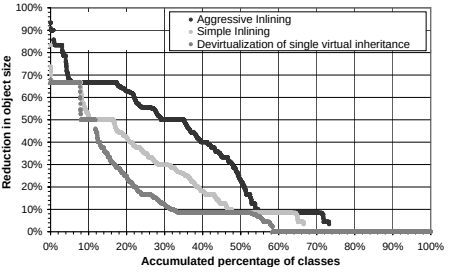
The three inlining techniques are compared in Fig. 17(a). We see that DEVIRT gives a consistent average saving of 10–20% in all object sizes. As explained earlier, WHO is nothing but CC after DEVIRT is applied. We argue that WHO$^+$ is much like CC$^+$, except that objects tend to be 10–20% smaller. Such saving may not seem enough to warrant transition to whole program analysis. Perhaps this is why many widely available C++ compilers do not apply whole program analysis to reduce object size. The shift to WHO can however be justified by the efficacy of the advanced optimization techniques.

It is important to note that whole program analysis for object layout does not mean in C++ (say) a batch compile of the whole program. All that is required is that object layout is decided in link time, and that appropriate fixup table is used to patch object-code files.

In small classes with no more than 30 compiler generated fields, SINLN$_{CC+}$ does not add much savings to DEVIRT. In larger classes, the additional savings by SINLN$_{CC+}$ are in the 15–35% range.



(a) Average efficacy of inlining techniques



(b) Accumulative distribution of efficacy of inlining techniques

Fig. 17: Average and accumulative efficacy of inlining in CC$^+$.

AINLN$_{CC^+}$ is significantly better than DEVIRT and SINLN$_{CC^+}$ in all object sizes, with the exception of classes whose size lies in the 21–30 range. This is unfortunate, since as observed in Sec. 4.2 one of the peaks in the distribution of classes lies exactly in this range. However, as we shall see below, bidirectional layout optimization techniques will be particularly effective in this range.

Another perspective at the comparison of the three inlining techniques is offered by Fig. 17(b). From the figure we can see for example that AINLN$_{CC^+}$ effects a 50% saving or more in 35% of all classes, SINLN$_{CC^+}$ makes a saving of at least 30% in 30% of all classes, while DEVIRT saves at least 25% of object size in 20% of all classes.

7 Bidirectional Object Layout

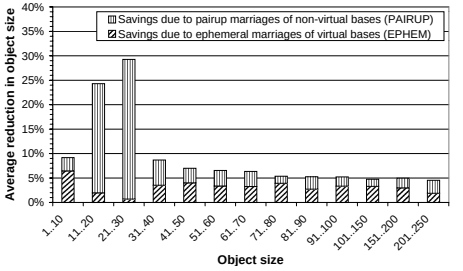
The bidirectional object layout idea is that all root classes and all other classes for which this is possible, are assigned a *directionality*. The directionality, which could be either positive or negative, is selected using a two-wise independent hash function. Since both positive and negative classes have their VPTR at offset zero, it is possible to "marry" a positive and negative bases in a derived class, whereby saving a VPTR. For example, in Fig. 1(a), if b is negative and c is positive, then a marriage could occur in a. Class a would then have *mixed* directionality. No VBPTRs can be saved by this technique.

7.1 Applying Bidirectional Layout after Inlining

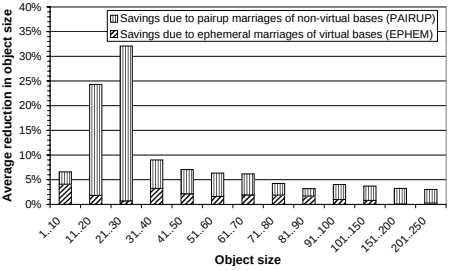
Fig. 19(a) is a scatter diagram showing the reduction in object size due to BIDIR applied after A-INLN was applied.

At first look, it appears that BIDIR$_{CC^+}$ is not a very effective technique, since, asymptotically, for large classes, it gives only about 5% reduction in object size. This however should be expected, recalling that BIDIR$_{CC^+}$ can only eliminate VPTRs. As we saw in Fig. 7(a) on average less than 20% of the size of classes whose size is greater than 80 is used by VPTRs. Thus, the $\approx 5\%$ observed saving is not in total disagreement with the theoretical prediction ([11]) of saving of around a quarter of VPTRs.

BIDIR$_{CC^+}$ has a stronger impact on smaller classes in which the weight of VPTRs is greater. We see in Fig. 19(a) that there is a cluster of classes, sized 20–25, in which the



(a) PAIRUP /SINLN$_{CC+}$ and EPHEM /SINLN$_{CC+}$



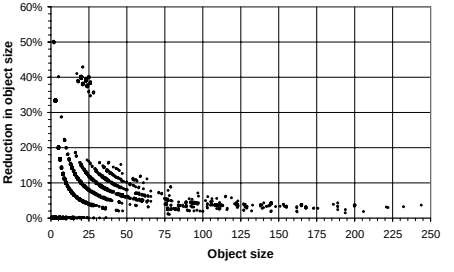
(b) PAIRUP /AINLN$_{CC+}$ and EPHEM /AINLN$_{CC+}$

Fig. 18: Average efficacy of PAIRUP and EPHEM.

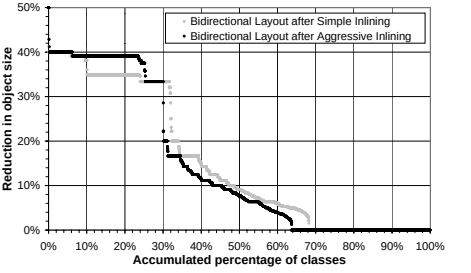
saving in object size is around 40%. Theses findings are confirmed by Fig. 18(b) shows the average savings due to BIDIR$_{CC+}$ in different object sizes. We see that for the peak in the distribution of class size located at the 21–30 bracket, BIDIR$_{CC+}$ makes a hefty saving of over 30%.

BIDIR is in fact a combination of two optimization techniques: PAIRUP, in which inlined and non-virtual parents are persistently married, and EPHEM in which virtual bases are ephemerally married. Fig. 18(b) shows also how the savings are broken down between these two sub-techniques.

It is quite surprising to see that even in large classes, not too much saving is owed to EPHEM. This is quite surprising in view of Fig. 3 which shows that there are many classes with a large number of virtual bases. The explanation is that the preliminary inlining optimization eliminated so many virtual bases that the optimization opportunities of EPHEM were severely restricted. This phenomena can also be seen by examining Fig. 18(a), which shows how the saving due to BIDIR are broken down between PAIRUP and EPHEM after SINLN$_{CC+}$, a less effective inlining technique.



(a) Efficacy of BIDIR /AINLN$_{CC+}$



(b) Accumulative distribution of efficacy of BIDIR /SINLN$_{CC+}$ and BIDIR /AINLN$_{CC+}$

Fig. 19: Efficacy and accumulative distribution of BIDIR /SINLN$_{CC+}$ and BIDIR /AINLN$_{CC+}$.

In comparing Fig. 18(a) and (b), we see that although BIDIR after S-INLN is slightly better than BIDIR after A-INLN, their performance is quite similar. However, when S-INLN was applied, EPHEM took a greater share of the total achievements of BIDIR.

Yet another comparison of between these two alternatives is given in Fig. 19(b). Again, we see that the two alternatives are quite similar: they both achieve about 30% savings in about 30% of all classes. The conclusion from this data is that inlining may hinder bidirectional optimization it is better to apply A-INLN and then BIDIR. This conclusion is confirmed by theoretical, probabilistic considerations which show that the expectation of the saving in VPTRs is decreased by inlining.

7.2 Hermaphrodite Object Layout

In plain bidirectional layout marriage opportunities may be missed due to unfortunate selection in the pseudo-random assignment of directionality to classes.

Going back again to Fig. 1(a), if the hash function assigned a positive directionality to both b and c, then no VBPTR could be saved in a. *Hermaphrodite* class is a class whose instances may use one of two different layouts: one using positive and the other with negative one. Initially, all root classes are hermaphrodite. When a class u inherits from h hermaphrodite parents, then if h is even, $h/2$ marriages occur, where appropriate directionalities are selected for each of the h subobjects.

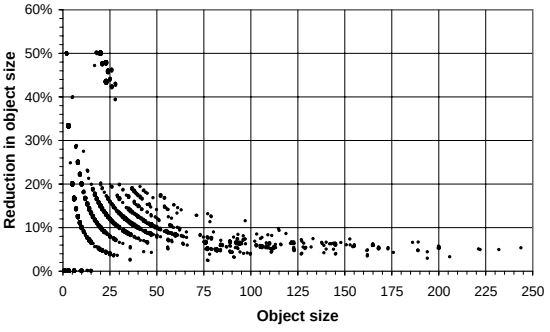


Fig. 20: Efficacy of H-BIDIR /AINLN$_{CC^+}$.

The class u has then mixed directionality, but there is only one possible layout for its objects. If h is odd, then $(h-1)/2$ marriages occur, fixing the directionality of $h-1$ subobjects. The remaining parent, v, remains hermaphrodite in u. Class u becomes hermaphrodite as well, where its instances of positive (respectively negative) directionality use a positively (negatively) directed u subobject.

All hermaphrodite classes have their VPTR at offset zero. However, in order to make a virtual function call, it must be possible to determine at run time the directionality of a specific instance, since the directionality of the VTBL is the same as the directionality of the object.[4] Since in many modern architectures, not all bits of a pointer are used, directionality could be stored as one of these bits, say LSB. Here is the pseudo-code for invoking a virtual function f using a pointer p to a hermaphrodite class.

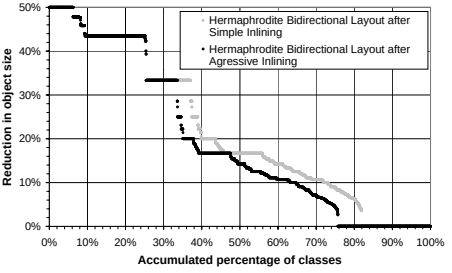
The implementation of this algorithm in machine code and its run time efficiency are very dependent on the instruction set. It could even be possible to develop dedicated hardware support. Conversely, with the absence of such support, this dispatch mechanism would bloat the code significantly. All these questions are beyond the scope of this paper. Instead, we are interested in the saving in per-object memory that this technique may

[4] A similar need arises for data members access. However, data members access could be restricted to member functions, and each hermaphrodite object could have also two versions for each of its member functions.

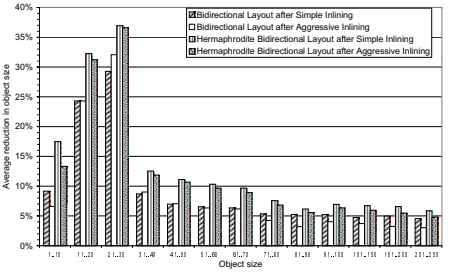
bring about. Fig. 20 shows the saving in object size due to H-BIDIR when applied after AINLN$_{CC+}$. The results are similar in appearance to Fig. 19(a), except that as expected, greater savings are demonstrated in all object sizes. For example, the cluster of classes of size 20–25 sees a 45% reduction in its size rather than 40%.

Fig. 21(a) compare the performance of HBIDIR$_{CC+}$ relative to the inlining algorithm that was applied prior to it. It appears, and in a much clearer way than in Fig. 19(b) that better inlining has adverse affects on the performance of a bidirectional layout. For example, if SINLN$_{CC+}$ was applied than 37% of all classes see a reduction of a third or more in their size, whereas only 33% of all classes see a similar reduction if AINLN$_{CC+}$ was applied.

```
VTBL ← *⌊p/2⌋
off ← offset(f)
if odd(p) then
   off ← −off
end if
call VTBL [off]
```



(a) Accumulative distribution of efficacy of H-BIDIR /SINLN$_{CC+}$ and H-BIDIR /AINLN$_{CC+}$



(b) Average efficacy of BIDIR /SINLN$_{CC+}$, BIDIR /AINLN$_{CC+}$, H-BIDIR /SINLN$_{CC+}$ and H-BIDIR /AINLN$_{CC+}$

Fig. 21: Accumulative and average efficacy of BIDIR and H-BIDIR in CC^{+}.

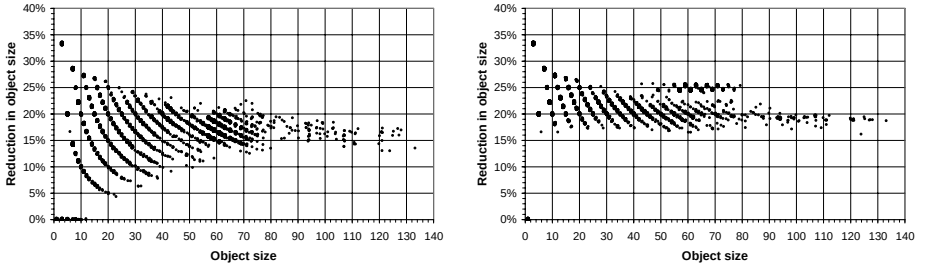
Finally, Fig. 21(b) compares the performance of BIDIR$_{CC+}$ and HBIDIR$_{CC+}$ when applied after SINLN$_{CC+}$ and AINLN$_{CC+}$, Even though HBIDIR$_{CC+}$ is better than BIDIR$_{CC+}$, the differences are not so big, except for the smaller classes, sized 1–10. The incurred dispatch cost of H-BIDIR could be justified in those applications which have a large number of classes of this kind, which is hard to optimize using any other technique.

7.3 Using Bidirectional Optimization in Separate Compilation Model

In contrast with inlining optimization, bidirectional object layout can be done without whole program analysis. It is therefore interesting to evaluate the efficacy of these algorithms in a separate compilation model. Since SEP^{+} seems to be impractical, SEP^{-} is used as baseline for this benchmarking.

The savings in object size due to BIDIR$_{SEP-}$ in are shown in Fig. 22(a). The figure demonstrates that there is a cluster of classes with savings in the 15–20% range. Further, saving is guaranteed for all classes with 13 or more compiler generated fields.

As Fig. 22(b) shows, the switch to hermaphrodite in SEP$^-$ raises a bit the range of typical savings.



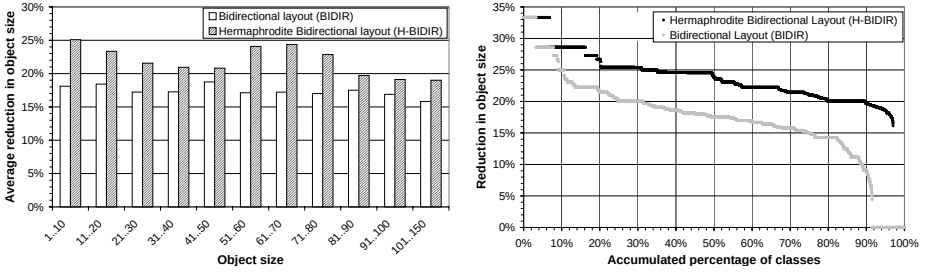
(a) BIDIR$_{SEP-}$ (b) HBIDIR$_{SEP-}$

Fig. 22: Efficacy of Bidirection Layout techniques in SEP$^-$.

Moreover, the figure demonstrates what can also be easily inferred from a pigeon hole principle: saving is guaranteed for all classes of size at least three.

Fig. 23(a) gives a side by side comparison of the efficacy of BIDIR and H-BIDIR in SEP$^-$ in SEP$^-$ for different object sizes. In no range the efficacy of H-BIDIR is bounded above by 25%. This bound is explained by our previous observation that in SEP$^-$, the number of VBPTRS is about the same as that of VPTRS, no savings in VBPTRS are made by bidirectional techniques, whereas at most half of all VPTRS can be saved. This meager saving of quarter in object size is far from bridging the gap between CC$^-$ and SEP$^+$ (Fig. 9)(a).

It is also worth noting that in no size bracket HBIDIR$_{SEP-}$ is better than BIDIR$_{SEP-}$ by more than 8%. The difference between the techniques is also highlighted in Fig. 23(b) showing the distribution of their savings in SEP$^-$.



(a) Average efficacy of BIDIR$_{SEP-}$ and HBIDIR$_{SEP-}$

(b) Accumulative distribution of efficacy of BIDIR$_{SEP-}$ and HBIDIR$_{SEP-}$

Fig. 23: Average and accumulative efficacy of Bidirection Layout techniques in SEP$^-$.

8 Summary

Table 8 summarizes the efficacy of the optimization techniques introduced in this paper. With aggressive lining we can achieve average savings of 32%, regardless whether the average is computed over all classes or just leaves. Applying bidirectional layout after inlining gives an additional average saving of 14.7%, which is very close to the efficacy of devirtualization. In applying the whole optimization suit, almost 50% of compiler generated fields can be eliminated. The last two lines in Table 8 are concerned with SEP^-. Bidirectional object layout optimization reduces object size in this case by more than 17%.

	Savings			
	Median		Average	
Technique	overall	leaves	overall	leaves
CC^+				
ETRANS	0.0%	0.0%	2.0%	1.9%
DEVIRT	8.5%	8.7%	13.9%	16.3%
S-INLN	0.0%	0.0%	6.0%	5.5%
A-INLN	0.0%	0.0%	16.0%	13.9%
BIDIR /S-INLN	9.1%	8.6%	15.2%	15.0%
BIDIR /A-INLN	7.7%	7.1%	14.7%	14.7%
H-BIDIR /S-INLN	16.7%	16.7%	21.8%	20.8%
H-BIDIR /A-INLN	14.3%	13.6%	19.7%	19.5%
SEP^-				
BIDIR	17.5%	17.9%	17.2%	17.4%
H-BIDIR	23.8%	23.5%	23.4%	23.3%

Table 8: The efficacy of all optimization techniques.

9 Conclusions

We studied the topological properties of large inheritance hierarchies which make use of MI. It was found that not very many classes make direct use of MI, and the ones that do have a relatively small number of parents. Further, the number of virtual bases is relatively small. Nonetheless, we found that the effects of MI can be seen throughout the hierarchy, and many classes have a large number of virtual bases. Despite this, hierarchy DAGs tend to share many of the properties of balanced trees.

The object size due to several layout strategies was investigated. Large classes with 50 and more compiler generated fields were found in significant portions even for CC^-, which seems to be the most efficient strategy in wide use. Even larger classes were found in CC^+, the strategy favored by most C++ compilers. Object sizes found in SEP^+ seemed so large to render this strategy impractical. On the other hand, SEP^- may suffer from timing problems due to the cost of casting to virtual bases, although we were unable to give an accurate estimate on the penalty in runtime in this strategy.

Concentrating on CC$^+$, we studied a variety of optimization techniques, grouped into two families: inlining and bidirectional layout. It was found that inlining is particularly effective for large classes, while bidirectional layout is more suitable for medium and small sized objects.

The use of hermaphrodite object layout, a new optimization technique contributed here, can improve on bidirectional layout by another 5–6% in general. However, hermaphroditing is especially effective in very small classes, where it guarantees saving in all classes having more than one VPTR or more than one *e*-VBPTR. It is not clear however what is the runtime cost of dispatch with hermaphrodite layout.

Some of the directions for future research include:

Repeated inheritance. Based on several independent reasons we assumed here that no repeated inheritance can occur. It would be interesting to study the extent to which repeated inheritance is used in practice, and if indeed it is found that there is more an occasional use of it, to try to optimize it. The main difficulty is that bidirectional layout is much less effective in the presence of repeated inheritance, and that inlining is only effective for virtual inheritance.

Instantiation frequency. Even though our working assumption was that classes are equally likely to be instantiated, the efficacy of optimization techniques was also studied for different object size. Sweeney and Burke [28] confirmed that there is a wide variety in the frequency of instantiation of different classes. It would be important to experiment with these optimization techniques in a sample of applications running with a sample of inputs.

Data member optimization. Non-compiler generated fields were not included in our study. It is important not only to compare the overhead due to compiler generated fields with actual object size, but also to apply our techniques to the optimization of ordinary data members. For example, bidirectional layout can easily be applied to expanded fields. For fields which use reference semantics one might be able to use concepts of ownership [21] and Balloon types [2] for the purpose of inlining.

Acknowledgments. We are grateful to Peter Sweeney of the IBM T.J. Watson research center for fruitful, stimulating and encouraging discussions.

References

[1] M. Akşit and S. Matsuoka, editors. *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In Akşit and Matsuoka [1].

[3] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual calls. In OOPSLA'96 [22].

[5] 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.

[6] Y. Caseau. An object-oriented deductive language. *Annals of Mathematics and Artificial Intelligence*, Mar. 1991. Special issue on deductive databases.
[7] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In Paepcke [23], pages 271–287.
[8] B. J. Cox. *Object-Oriented Programming - An Evolutionary Approach.* Addison-Wesley, 1986.
[9] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the 10^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 141–155, Austin, Texas, USA, Oct. 15-19 1995. OOPSLA'95, Acm SIGPLAN Notices 30(10) Oct. 1995.
[10] K. Driesen and U. Hölzle. The direct cost of virtual functions calls in C++. In OOPSLA'96 [22], pages 306–323.
[11] J. Y. Gil and P. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In *Proceedings of the 14^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 256–275, Denver, Colorado, Nov.1-5 1999. OOPSLA'99, Acm SIGPLAN Notices 30(11) Nov. 1999.
[12] R. Giladi and N. Ahituv. SPEC as a pefromance evaluation measure. *Computer*, 28(8):33–42, Aug. 1995.
[13] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In Paepcke [23], pages 394–410.
[14] Interactive Software Engineering. Ise eiffel compiler. See `http://www.eiffel.com`, 1999.
[15] A. Krall, J. Vitek, and R. N. Horspool. Efficient type inclusion tests. In *Proceedings of the 12^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–157, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, Acm SIGPLAN Notices 32(10) Oct. 1997.
[16] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In Akşit and Matsuoka [1], pages 128–145.
[17] M. A. Linton and D. Z. Pan. Interface translation and implementation filtering. In D. Lea, editor, *the 6th C++ Conference*, pages 227–236, Cambridge, MA, Apr. 1994. USENIX.
[18] 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.
[19] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice-Hall, 1988.
[20] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.
[21] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *Proceedings of the 12^{th} European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 158–185, Brussels, Belgium, July20–24 1998. ECOOP'98, Springer Verlag.
[22] OOPSLA'96. *Proceedings of the 11^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, California, Oct. 6-10 1996. Acm SIGPLAN Notices 31(10) Oct. 1996.
[23] A. Paepcke, editor. *Proceedings of the 8^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, USA, Sept. 26 - Oct. 1 1993. OOPSLA'93, Acm SIGPLAN Notices 28(10) Oct. 1993.
[24] M. Sakkinen. The darker side of C++ revisited. *Structured Programming*, 13:155–177, 1992.
[25] Standard Performance Evaluation Coroporation. SPECjvm98 documentation, release 1.0. Online version at `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`, Aug. 1998.

[26] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Mar. 1994.

[27] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[28] P. F. Sweeney and M. Burke. A methodology for quantifying and evaluating the space overhead in C++ object models. Technical Report RC21370, IBM T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA, Dec. 1998.

[29] D. Ungar and R. B. Smith. SELF: The power of simplicity. In N. K. Meyrowitz, editor, *Proceedings of the 2^{nd} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–241, Orlando, Florida, Oct. 4-8 1987. OOPSLA'87, Acm SIGPLAN Notices 22(12) Dec. 1987.