Implementing Statically Typed Object-Oriented Programming Languages

ROLAND DUCOURNAU
LIRMM – CNRS et Université Montpellier II, France

Object-oriented programming represents an original implementation issue due to its philosophy of making the program behaviour depend on the dynamic type of objects. This is expressed by the *late binding* mechanism, aka *message sending*. The underlying principle is that the address of the actually called procedure is not statically determined at compile-time, but depends on the dynamic type of a distinguished parameter known as the *receiver*. A similar issue arises with attributes, because their position in the object layout may also depend on the object's dynamic type. Furthermore, subtyping introduces another original feature, i.e. runtime subtype checks. All three mechanisms need specific implementations and data structures. In static typing, late binding is generally implemented with so-called *virtual function tables*. These tables reduce method calls to pointers to functions, through a small fixed number of extra indirections. It follows that object-oriented programming yields some overhead, as compared to usual procedural languages.

The different techniques and their resulting overhead depend on several parameters. First, inheritance and subtyping may be single or multiple and a mixing is even possible, as in Java and .Net which present single inheritance for classes and multiple subtyping for interfaces. Multiple inheritance is a well known complication. Second, the production of executable programs may involve various schemes, from global compilation, which implies the *closed-world assumption* (CWA) as the whole program is known at compile time, to separate compilation and dynamic loading, where each program unit is compiled and loaded independently of any usage, hence under the *open-world assumption* (OWA). Global compilation is well known to facilitate optimization.

This article reviews the various implementation techniques available in static typing and in the three cases of single inheritance, multiple inheritance, and multiple subtyping. This language-independent survey focuses on separate compilation and dynamic loading, as they represent the most commonly used and the most demanding framework. However, many works have been undertaken in the global compilation framework, mostly for dynamically typed languages but also applied to the Eiffel language. Hence, we also examine global techniques and how they can improve the implementation efficiency. Finally, mixed frameworks that combine open and closed world assumptions are considered. For instance, just-in-time (JIT) compilers work under provisional CWA, at the expense of possible recompilations. In contrast, we present an experimental compiler-linker, where separate compilation implies the OWA, whereas the whole program is finally linked under the CWA.

Categories and Subject Descriptors: D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; C#; Eiffel; Java; Prm; Theta; D.3.3 [**Programming languages**]: Language constructs and features—*classes and objects*; *inheritance*; D.3.4 [**Programming languages**]: Processors—*compilers*; *linkers*; *loaders*; E.2 [**Data**]: Data Storage Representation—*object representation*

General Terms: Languages, Measurement, Performance

Additional Key Words and Phrases: binary tree dispatch, casting, coloring, downcast, dynamic loading, genericity, late binding, linking, message sending, method dispatch, multiple inheritance, pointer adjustment, single inheritance, static typing, type analysis, separate compilation, virtual function tables

Author's address: R. Ducournau, LIRMM, 161, rue Ada – 34392 Montpellier Cedex 5, France

1. INTRODUCTION

The key feature of object-oriented programming is the fact that the program behaviour depends on the dynamic type of objects. This is usually expressed by the metaphor of message sending. Instead of applying a procedure or a function to an argument, a message is sent to an object—called the *receiver*, whereas the procedure or function is called a *method*—and the program behaviour, i.e. the code which will be executed, is determined by the receiver itself *at runtime*. In class-based languages, all proper instances of the same class share the same behaviour, hence message sending is interpreted according to the dynamic type of the receiver. From an implementation standpoint, it follows that the *static* procedural call of procedural languages must be replaced by some *dynamic* call, i.e. control flow jumps to an address extracted from the receiver itself. This is called *late binding*. In statically typed languages, late binding is generally implemented with tables, called *virtual function tables* in C++ jargon, and an object is laid out as an attribute table, with a header pointing to the class table. Method calls are then reduced to pointers to functions, through a small fixed number of extra indirections. A similar issue arises with attributes since their position in the object layout may depend on the object's dynamic type. Furthermore, subtyping introduces another original feature, i.e. runtime subtype checks. All three mechanisms need specific implementations and data structures that generally yield some overhead compared to procedural programming. This overhead depends particularly on inheritance; it is small with single inheritance, but multiple inheritance may increase it markedly.

This article describes the techniques that are commonly used for implementing object-oriented specific mechanisms, examines many alternatives, and evaluates and compares them. The scope of this survey is mainly restricted to: (i) *class-based* object-oriented languages (prototype-based languages like Self [Ungar and Smith 1987] will not be considered at all); (ii) *static typing* (languages like Smalltalk [Goldberg and Robson 1983] and Clos [Steele 1990] will not be considered, apart from rough comparisons); (iii) *separate compilation* (other compilation frameworks will be considered but not fully surveyed). Therefore, target languages are mostly C++ [Stroustrup 1998], C# [Microsoft 2001], Java [Gosling et al. 2005] and Eiffel [Meyer 1992; 1997]. Other languages exist, but these four must be used by more than 90% of object-oriented programmers, at least when only static typing is involved. Apart from these commonly used languages, new emerging ones might attract a large audience—e.g. Scala [Odersky et al. 2008]. In principle, language specifications should be independent of the implementation. However, in practice, most languages present a few features that closely depend on a precise implementation technique or compilation scheme. A contribution of this survey is to abstract implementation from language specifications. Hence, the techniques that underly current languages must appear as universal techniques that could apply to all languages, apart from general functional requirements like typing (static vs dynamic), inheritance (single vs multiple), compilation (separate vs global) and linking (static vs dynamic) that strongly constrain the implementation.

The targeted audience is twofold: (i) language designers and implementors should be primarily interested by the general survey and some in-depth analyses that might give new insights into the topic; (ii) programmers, teachers and students should be interested in this attempt at abstraction which could likely help them understand object-orientation, compare languages and analyze efficiency questions.

This survey was carried out within the framework of researches motivated by the following observation. In spite of its 40-year history, object-oriented programming is still hampered by a major efficiency issue in the *multiple inheritance* context, and this issue is worsened by *dynamic loading*. Due to the ever-increasing size of object-oriented class libraries, scalable implementations are crucial and there is still considerable doubt over the scalability of existing implementations. Thus, there is room for further research.

This article reviews implementation techniques, whether they are actually used in some language, described in the literature, or merely imagined as a point in the state space of possible implementations. A difficulty arises as most actual implementations are not described in the literature, either because they are not assumed to be novel, or for confidentiality reasons, because they are. Conversely, many techniques are theoretically described in the literature, without any known implementation. Thus, the schemes that we describe are more likely than real, but principles should not be too far from reality. Moreover, some techniques described here may be original, but this was not our primary goal.

1.1 Object-Oriented Mechanisms

This survey focuses on the core of object-oriented (OO) programming, that is the few features that rely on the object's dynamic type:

—object layout together with read and write accesses to attributes,

—method invocation and *late binding* in its most common single dispatch form, where the selection is based on one specific parameter, i.e. the receiver, which is bound to a reserved formal parameter called `self`[1];

—dynamic type checking which is the basis of constructs like *downcast*—indeed, though most considered languages are presumed to be type safe, all offer such constructs, which are needed for covariant overriding or for filling the lack of genericity such as in Java (up to 1.4);

—instance creation and initialization, through special methods called *constructors* in C++ and Java jargon.

A number of secondary features must also be considered. They are more or less explicit in the language specifications, but are all necessary. Some of them are not as trivial as they may seem. Inclusion polymorphism—i.e. the fact that an entity of a given static type may be bound to a value of a dynamic subtype—needs special attention as an object reference may depend on the static type of the reference; variable assignments, parameter passing and equality tests are instances of

[1] `Self` is the reserved word used in Smalltalk, and it corresponds to `Current` in Eiffel and `this` in C++ and Java. Here we adopt the Smalltalk usage, which is closer to the initial metaphor. `Self` is a reserved formal parameter of the method, and its static type is the enclosing class. This is also true in dynamic typing, although accesses to `self` are then not always statically type-checked.

this problem. Furthermore, some languages closely integrate primitive types in the class hierarchy, by defining *universal types* that are common supertypes of primitive types and object classes—for instance, `Any` in EIFFEL, `Object` in C# and JAVA 1.5. In this setting, the question of polymorphic use of primitive values, that is their binding to universally typed variables, deserves some examination. Uninitialized variables and attributes still represent a continual issue. Truly uninitialized references yield unrecoverable errors, e.g. *segmentation faults*, that are familiar to C++ programmers. Here we only consider implicit `null`-initializations, as in JAVA, which may lead to omnipresent, hence costly, tests. Looking for alternatives is worthwhile.

In contrast, we shall not consider non-OO features, that is all features whose specifications are based only on static types, hence whose behaviour is determined at compile-time:

—Non `virtual` methods in C++ and C#, or `static` variables and functions in JAVA, are specified and implemented in the same way as in non-OO languages; the only impact of object-orientation is the fact that classes are name spaces.

—*Static overloading* (à la C++, JAVA or C#) involves using the same name for different methods accessible in the same classes and distinguished by parameter types; ambiguities are statically solved by a selection which is equivalent to global renaming and must not be confused with late binding.

—*Protection* (aka *visibility*) rules the accessibility of some entity for the other ones; from the implementation standpoint, they are just static access rights to existing implementations. However, an exception will be made for SMALLTALK *encapsulation*, which reserves all attribute accesses for `self` and will be shown to be of interest for implementation.

1.2 Notations and Conventions

Regarding types and subtyping, we adopt a common point of view. We consider that classes are types and that class specialization is subtyping. Even though type theory distinguishes between both relationships, this is a common simplification in most languages. Type safety is assumed but static type checking, at compile-time, is beyond the scope of this article. According to the manguages, attribute and method *overriding* (aka *redefinition*) may be type-invariant or not and, in the latter case, it can be type-safe or unsafe; this is known as the *covariance-contravariance* problem; for the sake of simplicity, we consider that method signatures are invariant, but the variant case needs consideration and will be examined. Regarding values, we consider that a value is either a value of a primitive type or a reference—that is, the address of an object instance of some class. Thus we exclude the fact that an attribute value might be the object itself, as in C++ or with the EIFFEL `expanded` keyword. This simplifying assumption has no effect on the implementation.

Both classes and types are denoted by uppercase letters. The class specialization relationship is denoted by $\prec_{\mathrm{d}}$; $B \prec_{\mathrm{d}} A$ means that B is a *direct* subclass of A. One assumes that $\prec_{\mathrm{d}}$ has no transitive edges and its transitive (resp. and reflexive) closure is denoted by $\prec$ (resp. $\preceq$); the latter is a *partial order*. The terms *superclass* and *subclass* will be understood as relative to $\prec$, unless they are explicitly described as *direct*, i.e. relative to $\prec_{\mathrm{d}}$. A *root* is a class without any superclass. Subtyping is identified to $\preceq$. Finally, τ_s and τ_d, where $\tau_d \preceq \tau_s$, respectively denote the static

and dynamic types of an entity (e.g. a program expression); the static type is an annotation in the program text, possibly implicit for literals and `self`, whereas the dynamic type is the class instantiating the value currently bound to the entity.

Regarding *properties*, i.e. methods and attributes[2], we adopt the following terminology. A class *has* or *knows* a property if the property is *defined* in the class or in one of its super-classes. A class *introduces* a property when the property is defined in the class, not in its superclasses. Moreover, for taking inheritance and late binding into account, the term "property" does not denote a precise definition in a class. It denotes instead the abstract entity that could be called *generic property* in the model of CLOS *generic functions*, which has several *definitions* in different classes. As methods are commonly overridden, this distinction is important for them. On the contrary, multiple definitions of the same attribute are uncommon, so the confusion is harmless for attributes. In the literature on method dispatch, the SMALLTALK term of *method selector* is often used, whereas Zibin and Gil [2002] use *method family*. With static typing, a generic property must be identified with both its signature and the class introducing it. This accounts also for *static overloading*. This notion of generic property is very helpful for understanding multiple inheritance, but this is far beyond the scope of this article and interested readers are referred to [Ducournau and Privat 2008]. Indeed, implementation is actually not concerned by the actual definitions of methods, or by the inheritance problem itself, especially when multiple—namely, which method definition is inherited by some dynamic type? The only point is the existence of the corresponding generic property and an efficient way of calling the appropriate definition, whatever it is. Therefore, for the sake of simplicity, we shall use the terms *attribute* and *method* to denote generic properties and, if needed, we shall qualify them with *definition* to denote the declaration of the generic property in a given class.

1.3 Production Line of Executable Programs

Compilation Schemes. Implementation techniques are closely related to the way executable programs are produced. We shall distinguish between three main kinds of runtime production, that we will call *compilation schemes*:

—*separate compilation* coupled with *dynamic loading/linking* is a common paradigm with JAVA and .NET platforms;

—*separate compilation* and *global linking* may be the common naive view, e.g. for C++ programmers, even though the language and most operating systems allow for more dynamic linking;

—*global compilation*, including linking, is less common in production languages and EIFFEL is our main example, e.g. in the GNU compiler SMART EIFFEL (formerly known as SMALL EIFFEL) [Zendra et al. 1997; Collin et al. 1997].

[2]The object-oriented terminology is far from unified: C++ uses *member*, *data member* and *function member* instead of, respectively, *property*, *attribute* and *method*; EIFFEL *features* take the place of *properties*; *attributes* are often named *fields*, *instance variables* (SMALLTALK, JAVA) or *slots* (CLOS). In contrast, .NET *properties* denote a kind of attributes that are accessed through getter and setter methods with a field-access syntax.

These compilation schemes are not functionally equivalent from both the design and the programming standpoints. Separate compilation with dynamic loading corresponds to the *open-world assumption* (OWA), that best expresses the object-oriented philosophy, especially the software engineering requirement for reusability and extensibility. Each class should be designed and implemented while overlooking how it will be reused, for instance whether it will be specialized in single or multiple inheritance. In contrast, the other compilation schemes require a certain amount of *closed-world assumption* (CWA). The three compilation schemes are ordered, according to an increasingly closed world assumption, in such a way that all techniques available for an approach are also available for the following ones.

From the programmers' standpoint, separate compilation provides speed of compilation and recompilation together with locality of errors, and protects source code from both infringement and hazardous modifications. Hence, our primary interest is separate compilation but we shall also consider global techniques, especially when applicable at link-time, at least for comparison.

One may distinguish several tasks in compiling a class. (i) Code generation is common to all programming languages—it involves generating the target code for all procedures or functions, here methods. (ii) A second task consists of computing what we call here an *external schema* of the class, which must be used by all subclasses and *client classes*[3], in order to know which properties exist or are accessible in the considered class—this is a kind of interface, but it is not restricted to public properties and it includes method definitions, i.e. not the code but the existence of a definition. Without loss of generality, this schema can be thought of as an instance of some metamodel [Ducournau and Privat 2008]. (iii) The third task computes the data structures associated with classes, together with object layouts, by fixing the size of all tables and the offset of all entities. Task (ii) may be done by hand, as in C++ (`.h` header files), but it is better done by the compiler as in Java, and the schema must be extractable both from source and compiled code. Task (iii) can be done from the external schemata and the source code is not needed. The point is to determine whether task (i), which is the essential compilation task, does insert the offset values computed by task (iii) in the generated code. The obvious alternative is to use symbols as offsets, leaving the substitution of values for symbols to a further task, for instance linking, for which symbol resolution is a usual capability. The advantage of doing this at link-time is that many recompilations might be avoided, e.g. modifying the code of a method in a class should not require recompilation of subclasses and client classes.

With separate compilation, and dynamic loading even more, the corresponding tasks must be incremental, or at least should be efficiently recomputed at each load, which is not always possible and this point must be carefully examined. Dynamic loading is used to an increasing extent in modern platforms; it happens that most common implementation techniques are compatible with it. However, Java and .Net platforms are based on a runtime system—inherited from Lisp and Smalltalk—composed of a *virtual machine* coupled with an *interpreter* and a so-called *just-in-time* (JIT) compiler. Such an architecture has no effect on the un-

[3]Given a class A, a class B is a *client* of A if A, or a subclass of A, is used as a type annotation in the code of B. An alternative terminology is to say that B *imports* A.

derlying implementations, which are the same as with more classic runtime systems. However, JIT compilers are *adaptive* [Arnold et al. 2005]. Their behaviour can be thought of as separate compilation under temporary CWA. When a class is loaded, the compiler makes some assumptions that are currently verified—for instance some method calls are monomorphic—and compiles the code of the currently loaded class as if these assumptions were proven to be true. Later in the program execution, a new class loading may invalidate the assumptions—for instance the previously monomorphic call becomes polymorphic. Then the implied code must be recompiled. Therefore, adaptive compilers involve specific optimizations that apply at a different level, and they are generally beyond the scope of this survey. We shall thus focus on implementation techniques that do not require any recompilation.

1.4 Evaluating Efficiency

There are two basic criteria for efficiency, namely time and space. Time efficiency can be judged on average but constant-time mechanisms are ideal because they ensure an efficient worst-case behaviour. Space efficiency is evaluated by the amount of memory needed for runtime programs. Space and time efficiencies usually vary in opposite directions, although increasing the space occupation does not always improve time efficiency, as it also increases cache misses. So choosing a single criterion is impossible and a tradeoff is always needed.

Finally, run-time efficiency is the main goal but compile-time efficiency must not be overlooked; attention should be paid to NP-hard algorithmic optimizations.

1.4.1 *Space Efficiency.* Three kinds of memory resources must be considered. The dynamic part consists of the objects themselves, implemented as an attribute table, with one or more pointers to class tables—*garbage collection* must be envisaged for this part. The static part consists of data structures associated with classes, which are generally read-only and thus allocated in the code area, together with the code itself, where a given mechanism is handled by a sequence of instructions.

Static memory analyses have been carried out in the framework of global techniques which aim at compacting the large class-method dispatch matrix resulting from an injective numbering [Driesen et al. 1995; Ducournau 1997]. Similar analyses have not yet been performed for separate compilation and static typing. As for dynamic memory, some common techniques may have a large overhead. For instance, C++ subobject-based implementation yields many compiler-generated fields in the object layout. Thus, dynamic space also requires precise assessment.

Overall, we base our evaluation of space efficiency on statistics computed on benchmarks commonly used in the literature, together with a simple worst-case analysis. These benchmarks are reduced to an abstract description of class hierarchies, consisting of class schemata, hence without the code of methods and mostly independent of actual languages. A more accurate analysis of memory usage, cache and heap optimizations is beyond the scope of this survey.

1.4.2 *Time Efficiency and Processor Architecture.* With former classic processors, a measure of the time efficiency of a mechanism was the number of machine instructions it requires. The pipe-line architecture of modern processors, with instruction-level parallelism and branch prediction, makes this measure obsolete. In return, memory access and unpredicted branch cause multi-cycle latency. Thus

the time spent for one instruction is no longer one cycle, and the time measure must consider cycles. Therefore, the instruction number is mostly a space measure. Implementing method calls with direct access into method tables has long been considered optimal; the effective overhead vs. static function calls seemed unavoidable. However, the branching prediction of modern processors appears to have a crucial role in the method invocation efficiency. Currently, only conditional branchings are well predicted, and this is in favour of the technique, known as *inline cache* or *type prediction*, which involves comparing the actual type of receiver with an expected type, whose method is statically known. When the test is statistically well predicted for whole program execution, this technique is very efficient [Driesen et al. 1995; Zendra et al. 1997]. In contrast, table-based techniques might be considered out of date, because indirect branching is hard to predict. Nevertheless, several arguments oppose this thesis: (i) indirect branching prediction could be improved in future processors, thus putting both techniques on an equal level [Driesen 2001]; (ii) type prediction is not well adapted to separate compilation; (iii) the optimum certainly involves combining both approaches.

Anyway, we present here an evaluation of time efficiency based on a model of processors and on an intuitive pseudo-code, both borrowed from [Driesen and Hölzle 1995; 1996; Driesen et al. 1995; Driesen 2001]. Each code sequence is measured by an estimate of the cycle count, parametrized by memory `load` latency—L whose value is 2 or 3—and branching latency—B whose value may run from 3 to 15. In contrast, a `store` takes 1 cycle. Both latencies assume that the data is cached, i.e. a cache miss would add up to 100 cycles. For want of space, we shall not describe the code sequence of all the presented techniques, but interested readers can easily extrapolate them from the examples presented here (see Appendix D), and in related papers, e.g. in K. Driesen's work. Of course, this model provides only a rough evaluation, but more accurate than the instruction count, of the cost of each mechanism, with *all other things being equal*. A more objective evaluation should measure the execution time of benchmark programs compiled according to the different implementation techniques. Whereas such inter-language benchmarks are common in functional programming, they do not exist for OO languages. This is mostly due to the diversity of languages, programming styles and class libraries. Substantial evaluation and comparison work has been carried out in the framework of a single language, first for dynamically typed languages like Smalltalk, Self and Cecil, then for Java and .Net platforms. For instance, the SPECjvm98 (`http://www.spec.org/jvm98/`) and DaCapo [Blackburn et al. 2006] benchmarks are now widely used for Java. There are almost no comparisons between C++, Java and Eiffel implementations based on execution of actual programs. Some work has been done to compare the efficiency of different languages by measuring the execution time of artificial programs, automatically generated from an abstract language [Privat and Ducournau 2005], but the considered programs are too artificial to draw firm conclusions. More recently, Ducournau et al. [2009] described experimentation involving the compiler of an experimental language, Prm, compiling a real program according to a variety of implementation techniques and compilation schemes. These experiments are in progress and the results are still too partial to be included in this survey—however, they mostly confirm our abstract model.

1.5 Structure of the Article

Object-oriented implementation strongly depends on the inheritance and subtyping relationships, according to whether they are single or multiple. Our plan follows these distinctions. Section 2 presents the implementation principle in single subtyping (SST), i.e. single inheritance when all types are classes, that will serve as a reference throughout the article. Only the core of object-oriented programming is discussed here and the various notions are precisely introduced on the way. Section 3 describes the implementation principle with unrestricted multiple inheritance (MI), i.e. a C++-like, language-independent implementation based on subobjects. A possibly original optimization is described in the case where a subobject is empty. Section 4 then presents several alternatives to subobjects. The next section describes the median case of single class inheritance but multiple interface subtyping (MST), illustrated by Java and .Net languages. Some applications to multiple inheritance and the special case of *mixins* are examined. Whereas the previous sections only consider separate compilation and open world assumption, Section 6 proposes a short survey of the techniques available in more global frameworks, under the closed-world assumption. Coloring, type analysis, tree dispatch and link-time generation of dispatch code are examined. The article ends with a conclusion, some prospects, and an index of all acronyms used.

Complementary materials are gathered in electronic appendices. Appendix A examines some variants of the subobject-based implementation, especially C++ *non-virtual inheritance* implementation (NVI) and the so-called VBPTRs. Appendix B briefly examines a few related topics like polymorphic primitive types, generics and garbage collection. Appendix C presents space statistics on common large-scale benchmarks. A last appendix recalls Driesen's computational model.

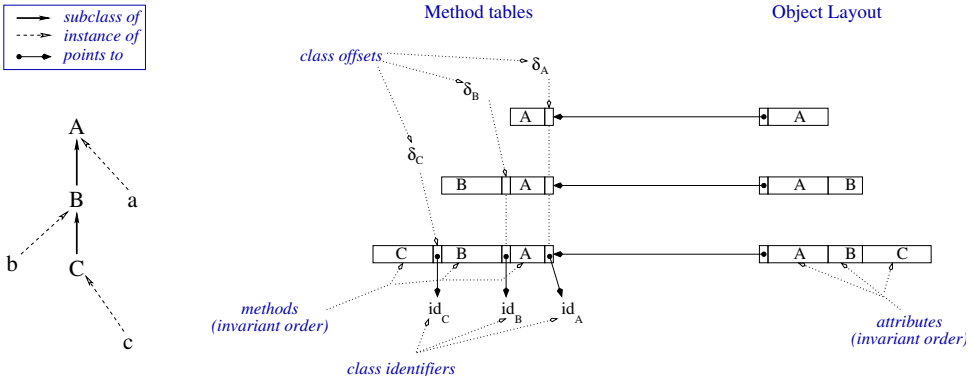
2. SINGLE INHERITANCE AND SUBTYPING (SST)

This section introduces the problem of object-oriented implementation in the simplified context of *single subtyping*, which implies that types can be identified with classes and that each class has at most one superclass. Although pure SST languages are not common, this implementation is the basis of most implementations, in both Java without interfaces (Section 5, page 30) and C++ when restricted to single and *non-virtual* inheritance (Appendix A.1).

Any implementation must satisfy some invariants for the representation of objects which characterize the whole implementation and make it work. Without loss of generality, they concern both *reference* and *position*. The invariant of reference must specify where an object is pointed to and how a reference (i.e. a method parameter or returned value, a local variable, an attribute) on an object behaves with respect to the static type of the reference. Another invariant must specify the position of a target inside the object representation. In this article, we shall present invariants that are enforced by the main implementations and we shall discuss their impact and consequences on the overall implementation of the languages.

2.1 Principle

Single subtyping provides an intuitive implementation. For a root class, the object layout is a simple array of attributes with a header pointing at the method table,



A small class hierarchy with 3 classes A, B and C with their respective instances, a, b and c. Method tables—including Cohen's display—(left) and object layout (right) in single subtyping. This diagram respects the following conventions. In method tables (left), A (resp. B, C) represents the set of addresses of methods introduced by A (resp. B, C). Method tables are drawn from right to left to reduce edge crossing. In object layouts (right), the same convention applies to attributes but the tables are drawn from left to right. Solid lines are pointers.

Fig. 1. Single-subtyping implementation

which is a simple array of method addresses. The subclass tables are simply obtained by adding newly introduced methods and attributes at the end of (a copy of) the direct superclass tables (Figure 1). The two straightforward invariants that characterize this implementation are the basis for constant-time access.

Invariant 2.1 (Reference). *Given an object, a reference to this object does not depend on the static type of the reference.*

Invariant 2.2 (Position). *The position of each attribute or method p is determined by an offset, denoted δ_p, that is invariant by inheritance and unambiguous, hence unique among all attributes or methods known by the same class. Conversely, given a static type, the position of an attribute (resp. method) in the object layout (resp. method table) is invariant with respect to the receiver's dynamic type.*

Thus, standard SST implementation is characterized by an absolute invariance with respect to static types. This enhances the basic semantics of object orientation, which states that the dynamic type is the object essence and that static types are pure contingency. Moreover, it represents ideal *type erasure*, as static types can be erased once the program has been type-checked.

Omitting parameter passing, which is done in the same way as in non OO languages, method calls are then compiled into a sequence of three instructions:

```
load [object + #tableOffset], table
load [table + #methodOffset], method                    2L + B
call method
```

where `object` is a register that holds a reference on the receiver, `#tableOffset` is a constant, usually 0, and `#methodOffset` is the δ_p value. The whole sequence takes $2L + B$ cycles. Attribute accesses are as immediate as for a record field:

```
load [object + #attributeOffset], attribute                    L
```

The conjunction of single inheritance and static typing avoids what we call *introduction overloading*, that is the fact that two properties with the same name might be *introduced* in two $\prec$-unrelated classes, hence without being *defined* in any common superclass. With static typing (and of course without multiple inheritance) the two occurrences are no more related than with *static overloading*. Indeed, any method call in the code can be statically and unambiguously assigned to one of the two methods. Therefore, computing the tables, i.e. method and attribute offsets δ_p, is straightforward—the result is both sound and optimal since no offset conflict can occur between inherited properties. Thus, if p is the last allocated method (resp. attribute), at offset δ_p, the offset for the next introduced method (resp. attribute) q will be $\delta_q = \delta_p + 1$, without any need to check that this new offset is really free. This algorithm is a special and easy case of coloring heuristics (Section 6.3, page 39).

Furthermore, each access to an attribute or a method of an object (`object` in pseudo-code examples) must be preceded by comparing `object` with the `null` value, as local variables and attributes may be left `null`-initialized. Assuming that the failure case, which must signal an exception, is shared, then this adds two instructions and cycles per access (see also Section 2.5, page 15).

2.2 Instance Creation

Eventually, instance creation amounts to: (i) allocating a memory area according to the number of attributes, (ii) assigning the method table address at `tableOffset`, (iii) calling a method for initializing attributes (improperly called a *constructor*). The (i-ii) stages are usually static, as the instantiated class occurs as a constant in the code. Instantiating a formal type (with genericity) or a virtual type might, however, require an actual method for instance creation. Regarding the initializing method, this is generally a standard method, ruled by late binding[4]. The question of uninitialized attributes may be dealt with by generating some assignments to `null` at compile time. A general alternative involves copying (aka cloning) a precompiled prototype of the instance.

2.3 Casting and Subtype Testing

2.3.1 *Principle.* Besides method invocation and attribute access, subtype testing is the third original mechanism that must be addressed by all object-oriented implementations. Given an entity x of a static type C (the *source*), the programmer or compiler may want to check that the value bound to x is an instance of another type D (the *target*), generally before applying to x an operation which is not known by C. Therefore, the mechanism is twofold: the test itself, which can be thought of in terms of success and failure; and a conditional static side-effect, called a *cast*, which allows the compiler to consider that the reference to x has type D. Most statically typed languages put the cast in the foreground, but the essence of the mechanism lies in the test.

More generally, the word *cast* is commonly used in reference to mechanisms close to type coercion, from a *source* to a *target* type. Among its various interpretations, two are of interest here, as they concern the relationship between static and dynamic

[4]Unlike C++, where all calls in a constructor are static, as if `this` were not polymorphic. It would be warranted by the fact that the language does not provide any implicit `null`-initialization.

types. Source and target types are then related by subtyping. The aforementioned mechanism associated with subtype testing is called *downcast*, since the target is generally a subtype of the source. *Downcast* is performed through special syntactic constructs like `dynamic_cast` in C++; parenthesized syntax (a C syntax which must not be used in C++!) in JAVA and C#; `typecase` in THETA [Liskov et al. 1995]; or *assignment attempts* in EIFFEL. In JAVA, exception `catch` clauses involve implicit `typecase`. *Downcast* uses may be justified by the fact that covariant models are implemented in type-safe languages. They are also common in JAVA because of its lack of genericity (up to version 1.5). Downcast failures may be treated in several ways, either by signalling an exception (JAVA, C++ for references only), returning a null value (EIFFEL, C++ for pointers only), or by jumping to the next case (`typecase`). Downcasts can also be replaced by a boolean operator, like JAVA `instanceof`, or a reflective method—however, this only ensures the underlying test.

In constrast, *upcast* is often called *implicit cast* because it requires no particular syntax. It simply involves a polymorphic assignment (or parameter passing) $x := y$, when the static type of x (resp. y) is X (resp. Y) and Y is a proper subtype of X ($Y \prec X$). Such a mechanism should have no name as it is conceptually vacuous—this is pure inclusion polymorphism—but its implementation may be non-trivial. Upcasts may also be useful for disambiguating static overloading.

Besides these two opposite directions, casting may also be *static* or *dynamic*[5], according to whether the target type is statically known or not. Explicit syntactic constructs are always static casts as the target type is a constant of the construct. Reflection provides means for dynamic casts, such as the `isInstance` method in JAVA. Furthermore, some features may need dynamic casts; for this the target type has to be reachable from the method table of the considered object.

2.3.2 *Casting in Single Subtyping.* As references to objects do not depend on static types, upcast is no longer relevant in the implementation, which is trivial, as is the concept. Downcast reduces to subtype testing, with two simple classic implementations.

Cohen's Display. The first technique has been described by Cohen [1991] after the "display" originally proposed by [Dijkstra 1960]. It consists of assigning an offset to each class in the method tables of its subclasses—the corresponding table entry must contain the class identifier. An object is an instance of a class C iff the object method table, noted tab_{τ_d}, contains the identifier id_C at offset δ_C:

$$\tau_d \preceq C \Leftrightarrow tab_{\tau_d}[\delta_C] = id_C \tag{1}$$

Class offsets are ruled by the same position Invariant 2.2 as methods and attributes.

The detailed implementation can vary in several ways: (i) the table can be separated from or inlined in the method table; (ii) if it is separate, its size can be variable (depending on each class), uniform (the same for all classes, but resizable) or even fixed (uniform and non-resizable); and (iii) the test may require a bound-check. Space optimizations exist, based on a variable-length encoding of class identifiers, which are no longer absolute but relative to classes with the same depth. Overall,

[5]These terms are unrelated to the C++ operators `static_cast` and `dynamic_cast`—both are *static*.

the most efficient implementation, that is somewhat new, consists of inlining Cohen's display in the method table, while avoiding bound checks by allocating the method tables in dedicated memory areas. Interested readers are referred to the discussion in [Ducournau 2008].

This first technique is simple and works well in separate compilation and dynamic loading, i.e. it is time efficient though not space optimal. It has been widely reused by different authors, e.g. [Queinnec 1998; Alpern et al. 2001b].

Schubert's Numbering. The second technique is due to Schubert et al. [1983] and also called *relative numbering*. It is time efficient and, in practice, space optimal but it has a twofold drawback, i.e. it does not generalize to multiple inheritance, at least in constant time, and it is not incremental, thus is less compatible with dynamic loading. It is a double class numbering, denoted n_1 and n_2: n_1 is a preorder depth-first numbering of the inheritance tree and n_2 is defined by $n_2(C) = \max_{D \preceq C}(n_1(D))$. Then:

$$\tau_d \prec C \Leftrightarrow n_1(C) < n_1(\tau_d) \leq n_2(C) \tag{2}$$

Only two short integers must be stored in the method table and the first one (n_1) can serve as a class identifier. For the test (2), $n_1(\tau_d)$ is dynamic, whereas $n_1(C)$ and $n_2(C)$ are static when the cast is static—they can be compiled as constants. When used in a dynamic loading framework, this technique requires some recomputation each time a class is loaded. As the complexity is linear in the class number, this is not a computational problem. Full recomputation can even be avoided in several ways, by lazy recomputation [Palacz and Vitek 2003] or pre-booking holes in the class numbering. But, in the worst case, a full recomputation is required. Moreover, $n_1(C)$ and $n_2(C)$ need now memory access, as in a dynamic cast. The overall efficiency is thus lower than that of Cohen's display (Appendix D.1, page App–24).

2.3.3 *Conclusion on Subtype Testing.* Type checking remains an active topic of research, even in single subtyping [Raynaud and Thierry 2001; Gil and Zibin 2005], but no other technique is as simple as Cohen's display and Schubert's numbering. Overall, we shall favor Cohen's display because of its better compatibility with dynamic loading. It also represents a general paradigm and generalizes to multiple inheritance in several ways. Therefore, hereafter "SST implementation" will denote the implementation presented in Section 2.1, coupled with Cohen's display.

2.4 Evaluation

This intuitive SST implementation provides a reference—one cannot expect to do better without specific optimizations. The techniques required for multiple inheritance or multiple subtyping will be compared with this reference, for both time and space efficiency.

All three mechanisms are time-constant. Moreover, time efficiency is optimal as everything is done with a single indirection in a table. Apart from attribute initialization, instance creation is also time-constant. Dynamic space efficiency is also optimal—object layout is akin to record layout, with the only overhead being a single pointer to class method table. Method tables depend only on object dynamic types. Overall, they occupy a space equal to the number of valid class-method pairs, which is the optimal compactness of the large class-method dispatch matrix

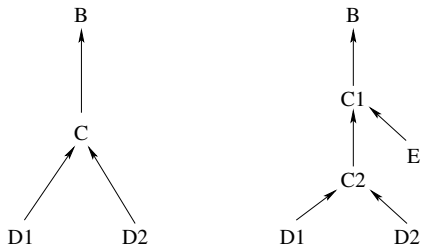


Fig. 2. Abstraction—C is split into C_1 and C_2 in order to define E.

often considered for constant-time techniques in global compilation (Section 6.2, page 37). Let M_C denote the number of methods known (defined or inherited) by a class C, then the total method table size is $\sum_C M_C$. Cohen's display adds exactly the cardinality of the specialization relationship, $|\preceq|$.

Production of Runtimes. This implementation is also incremental and fully compatible with dynamic loading. Class recompilation is needed only when the schemata of superclasses or imported classes are modified. Offset computation may be done at compile-time or delayed at load/link time and left to a quite general symbol resolution mechanism (Section 1.3, page 6). Only class identifiers must be computed at load/link time and not at compile time, but it can be done by the compiled code itself and does not require a specific linker or loader.

Space Linearity. In the SST implementation, the total table size is roughly linear in the cardinality of the specialization relation, i.e. linear in the number of pairs (x, y) such that x is a subclass of y ($x \preceq y$). Cohen's display uses exactly one entry per such pair, and the total size is linear if one assumes that methods and attributes are uniformly introduced in classes (see Appendix C.5, page App–21). Accordingly, the size occupied by a class is also linear in the number of its superclasses.

More generally, linearity of the total table size in the number of classes is actually not possible since efficient implementation requires some compilation of inheritance, i.e. some superclass data must be copied in the tables for subclasses. Therefore, usual implementations are linear in the size of the inheritance relationship. In the worst case (i.e. deep rather than broad class hierarchies), they are thus quadratic in the number of classes. In contrast, the C++ implementation is, in the worst case, cubic in the number of classes (Section 3, page 17). The fact that it is not possible to do better than linear-space is likely a consequence of the constant-time requirement. Indeed, Muthukrishnan and Muller [1996] propose an implementation of method invocation with $\mathcal{O}(N + M)$ table size, but $\mathcal{O}(\log \log N)$ invocation time, where N is the number of classes and M is the number of method definitions.

Abstraction Cost. From the design standpoint, it is interesting to examine the cost of abstraction as it enhances reusability. Designing a class hierarchy goes through successive specialization (adding a new subclass) and generalization steps. Generalization consists of splitting an existing class C into two related classes C_1 and C_2, such that $C_2 \prec C_1$, and $\forall X, C \prec X \Rightarrow C_1 \prec D$ and $X \prec C \Rightarrow X \prec C_2$ (Fig. 2). As C_2 takes the place of C, generalization does not change the behaviour

of existing programs—it only opens the door to further specialization of C_1. Generalization often yields abstract classes that have no proper instances and are used only for sharing code. These abstract classes entail no overhead at all in this implementation. More generally, when splitting a class, one does not add any dynamic overhead, only a small static space overhead if one does not know that the generalized class (C_1) is abstract. This point indicates that reusability can be increased without penalizing current programs.

2.5 Basic Optimizations

On the basis of this simple implementation of method invocation, attribute access and subtyping tests, two classic optimizations of programming languages may improve the resulting code, even in separate compilation.

Intra-procedural data flow analysis may: (i) avoid useless `null` checks, (ii) detect *monomorphic* cases, when τ_d is known at compile-time—method calls are then static function calls, without table accesses, and downcasts are statically solved. For instance, consider the following code:

```
1   x : A          // x = null
    ....
2   x := new B     // B ≺ A
    ....
3   x.foo()
4   x.bar()
```

According to the control flow of this program, the call to `foo` might be monomorphic, hence replaced by a static call to the `foo` method in B. If it is not—e.g. because of the code between lines 2-3—the `null` check might at least be avoided. Finally, if line 2 is removed, the `null` check is useless in the call to `bar`, even if it is needed for `foo`, as `x` has already been proven to be different from `null`. Languages also provide syntactic features, like keywords `final` in JAVA, `sealed` in C# and `frozen` in EIFFEL, that allow the programmer to express that a class cannot be specialized, or that a method cannot be redefined. To the detriment of reusability, these keywords provide easy intra-procedural detection of some static calls.

Inlining is another common optimization of procedural languages—it involves copying the code of the callee in the caller, for instance when the callee is either small or not often called. With OO languages, inlining can only apply to static calls, e.g. to monomorphic calls, and with separate compilation, it can only apply to methods whose source code is known, hence defined in the current code unit. Alternatively, the code to be inlined must be included in the *external schema* of the unit, as in C++ `.h` files.

Despite their restricted usage, both optimizations may have a significant effect, as the situation shown in the previous example is quite frequent.

2.6 First Methodological Conclusions

Besides this first evaluation, several conclusions can be drawn from the SST implementation that can serve as conceptual and methodological guides in our study.

2.6.1 *Grouping and Prefix Condition.* The SST implementation satisfies what we call the *prefix condition*. If $B \prec A$, then the implementation of A forms a *prefix*

Table I. Mechanism equivalence

technique	method	attribute	subtyping
SST	SST	SST	[Cohen 1991]
interval containment	[Muthukrishnan and Muller 1996]	SST	[Schubert et al. 1983]
coloring	[Dixon et al. 1989]	[Pugh and Weddell 1990]	[Vitek et al. 1997]
type slicing	[Gil and Zibin 2007]	AS	[Gil and Zibin 2005]
perfect hashing	[Ducournau 2008]	AS	[Ducournau 2008]

The table presents some equivalences between all three mechanisms. AS stands for *accessor simulation*, Section 4.1, page 26. Perfect hashing and coloring are respectively examined in Sections 4.2, page 27, and 6.3, page 39. Type slicing is a generalization of interval containment to multiple inheritance that we do not further consider.

in the implementation of B. As a corollary, attributes and methods are *grouped* according to their introduction class. No other implementation satisfies it for all pairs of $\prec$-related classes, but we shall see that this condition can provide a good opportunity for optimizations when two classes satisfy it.

2.6.2 *Mechanism Equivalence.* The three mechanisms that we consider—namely method invocation, attribute access and subtype testing—would seem to be equivalent, as they can be reduced to each other.

attributes ⇒ methods. Method tables are object layout *at the meta-level*, as methods can be considered as *shared immutable attributes*. Apart from memory-allocation considerations, their underlying implementations are thus equivalent.

methods ⇒ attributes. Conversely, an attribute can be read and written through dedicated *accessor* methods—hence, attribute access can always be reduced to method invocation (Section 4.1, page 26).

methods ⇒ subtype test. An interesting analogy between subtype tests and method calls can also be drawn from Cohen's display. Suppose that each class C introduces a method `amIaC?` which returns `yes`. In dynamic typing, calling `amIaC?` on an unknown receiver x is exactly equivalent to testing if x is an instance of C; in the opposite case, an exception will be signaled. In static typing, the analogy is less direct, since a call to `amIaC?` is only legal on a receiver statically typed by C; this is type safe but quite tautological. However, subtype testing is inherently type unsafe and one must understand `amIaC?` as a pseudo-method, which is actually not invoked but whose presence is checked. The test fails when this pseudo-method is not found, i.e. when something else is found at its expected position. This informal analogy is important—it implies that one can derive a subtype testing implementation from almost any method call implementation. We actually know a single counter-example, when the implementation depends on the static type of the receiver, as in subobject-based implementations (Section 3).

subtype test ⇒ methods. Conversely, when a subtype testing implementation relies on the search of a class ID in the data structure, a method invocation technique can easily be derived from it, by (i) grouping methods by introduction classes, (ii) associating with each class ID the offset of the corresponding method group. Hence, all subtype testing implementations that are somewhat similar to Cohen's display provide a method invocation solution. When the class ID is less directly encoded in the data structure, there are still ways of deriving a method invocation technique, as with interval containment [Muthukrishnan and Muller 1996] which derives from

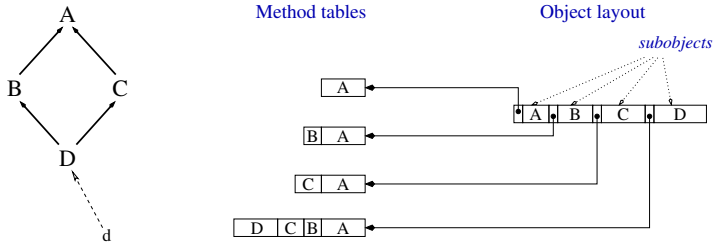


Fig. 3. Object layout and method tables with multiple inheritance: an instance d of class D is depicted. The diagram follows the same convention as in Figure 1.

Schubert's numbering.

Table I depicts the equivalence between a few techniques.

3. MULTIPLE INHERITANCE (MI)

Multiple inheritance complicates implementation to a considerable extent, as Ellis and Stroustrup [1990, chapter 10] and Lippman [1996] demonstrate for C++. This section presents a plausible implementation of this language while remaining language independent. Variations will be discussed in Appendix A.

3.1 Principle

With both separate compilation and MI, there is no way to maintain the invariants of position and reference that characterize single subtyping. Consider the two classes B and C in Figure 3. With the SST implementation, methods and attributes that are respectively introduced by B and C would occupy the same offsets. Hence, these properties would collide when the common subclass D is defined. We shall see further how to keep reference invariance, by giving up the constant-time requirement (Section 5.4, page 34) or the open-world assumption (OWA), i.e. separate computation of attribute and method offsets (Section 6.3, page 39). We now examine which invariants can allow an implementation to directly access the desired data in the object representation under the OWA.

3.1.1 *Object Layout and Method Tables.* The main idea of this implementation is to relax both invariants. The object layout now consists of subobjects, one for each superclass of its class (i.e. for each static type τ_s that is a supertype of its dynamic type τ_d) and each subobject is equipped with its own method table (Figure 3). Reference invariance is first replaced by the following:

Invariant 3.1 (Type-dependent reference). *Any reference whose static type is T is bound to the single subobject corresponding to T. Subobjects of different static types are distinct.*

Position invariance is then relaxed in different ways for attributes and methods.

Invariant 3.2 (Private attribute position). *Each attribute a has an offset, denoted δ_a and invariant with respect to the dynamic type, that determines the attribute position in the context of the static type* introducing *the attribute.*

Table II. Method tables for Figure 3 classes, according to static and dynamic types

type static → / ↓ dynamic	A	B	C	D
A	A	—	—	—
B	A	A B	—	—
C	A	—	A C	—
D	A	A B	A C	A B C D

The last line corresponds to Figure 3 tables. In each table, the class names (A, B, C, D) stand for the methods introduced by the class. Though method ordering for a given static type is arbitrary, it is convenient to group them per introduction class, with an invariant ordering in the group. Hence, for the same static type (column), the tables are isomorphic, i.e. same method ordering, but differ by their contents (method addresses), whereas, for the same dynamic type (row), isomorphic pieces contain the same addresses but different shifts. In this example, for each pair of $\prec$-related static types—apart from C and D, since AC is not a prefix of ABCD—the method tables verify the *prefix condition* which is needed for *empty subobject optimization* (Section 3.3, page 23).

Invariant 3.3 (Type-dependent method position). *Given a method m and a static type τ_s which* knows *m, the method position is determined by an offset denoted $\delta_m^{\tau_s}$ and invariant with respect to the receiver's dynamic type.*

Method (resp. attribute) offsets are no longer absolute, but relative to the current static type (resp. the introduction type). A subobject consists only of the attributes *introduced* by the corresponding static type, whereas its method table contains all methods *known* by it, with values (addresses) corresponding to methods inherited by the dynamic type. Thus, two proper instances of different classes do not share any method table of their common superclasses—some of these tables are isomorphic but do not contain the same addresses (Table II, page 18). For a given static type, the method ordering does not matter. It does, however, make sense to group methods by introduction classes, as in the figures, because this may simplify the compiler code, but this organization has no effect on efficiency. Moreover, the *prefix condition* (Section 2.6.1, page 15) can be required for further optimization (Section 3.3, page 23). Hence, a preferred method ordering for a class would be to extend the method ordering of at least one of its direct superclass. The dissymmetry between attributes and methods that appears in this implementation somewhat refutes our previous conclusions (Section 2.6, page 15). Whereas both kinds of property are conceptually analogous, the implementation distinguishes *introduced* attributes and *known* methods. The explanation is that methods are *shared immutable attributes*. Being immutable, they can be copied without changing the program semantics. Hence, copying methods is an optimization which cannot be applied to attributes, which are mutable. Appendix A.2 describes a more symmetrical, but less efficient, implementation.

3.1.2 *Method Call and `self` Adjustment.* Invariant 3.1 requires recomputation of the value of `self` for each method call where the static type τ_s of the receiver in the caller is different from its static type in the callee, that is the class W that defines the selected method (Figure 4). The position of subobject W with respect to subobject τ_s is required. It is denoted $\Delta_{\tau_s,W}$, and we call this relative position between subobjects a *shift*. Shifts and notations always imply a dynamic type, i.e. $\Delta_{T,U}$ is the distance between the T and U subobjects in the implementation of $\tau_d \preceq T, U$. Thus method tables have double entries for each method, an address

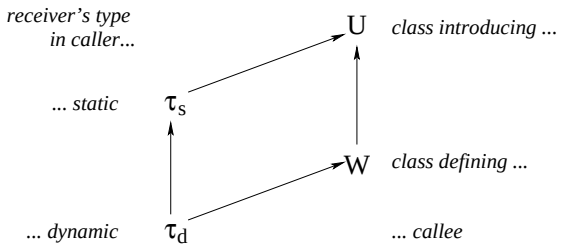


Fig. 4. Receiver types in a method call—a shift $\Delta_{\tau_s,W}$ is needed

and a *shift* (Figure 13, page App–5). Overall, method calls are compiled into the following sequence, where instructions added to SST (see page 10) are italicized:

```
load [object + #tableOffset], table
load [table + #methodOffset+1], delta
load [table + #methodOffset], method                    2L + B + 1
add object, delta, object
call method
```

The `object` register increment must not be considered as a change in the enclosing environment. The cycle count is only $2L + B + 1$ because the extra `load` runs in parallel—see Appendix D.2, page App–25.

Instead of putting shifts in the tables, an alternative is to define a small intermediate piece of code, called *thunk*[6] by Ellis and Stroustrup [1990] or *trampoline* by Myers [1995], which shifts the receiver before jumping to the method address:

```
add object, #delta, object
jump #method
```

The advantage is that `#delta` (i.e. $\Delta_{\tau_s,W}$) is now an immediate value and the call sequence is the same as with SST. Thus, one table access is saved for an extra direct branching and the thunk with the null shift is the method itself. A thunk must be generated when compiling τ_d, for each such pair (τ_s, W). Therefore, the thunk may be understood as a method redefinition which only calls `super`. However, a different redefinition is required for each static type τ_s.

[6]According to Lippman [1996], *thunk* would be *Knuth* spelled backwards. However, most people agree that the term originates from ALGOL `call-by-name` parameter passing:

Historical note: There are a couple of onomatopoeic myths circulating about the origin of this term [i.e. thunk]. *The most common is that it is the sound made by data hitting the stack; another holds that the sound is that of the data hitting an accumulator. Yet another suggests that it is the sound of the expression being unfrozen at argument-evaluation time. In fact, according to the inventors, it was coined after they realized (in the wee hours after hours of discussion) that the type of an argument in Algol-60 could be figured out in advance with a little compile-time thought, simplifying the evaluation machinery. In other words, it had "already been thought of"; thus it was christened a 'thunk', which is "the past tense of 'think' at two in the morning"* (`http://www.retrologic.com/jargon/T/thunk.html`). `http://www.webopedia.com/TERM/T/thunk.html` and `http://en.wikipedia.org/wiki/Thunk` report similar origins. The precise meaning is slightly different here. Anyway, a thunk may be defined as a small piece of code, or a stub function, used for parameter adjustment.

3.1.3 *Virtual Inheritance in* C++. The principle of this multiple inheritance implementation is that of C++. However, C++ is all the more complicated because it offers some features which aim at reducing MI overhead but reduces the code reusability. For instance, this implementation is obtained in C++ by annotating each superclass with the keyword `virtual` (this use of `virtual` must not be confused with its use for function annotation). Moreover, beyond implementation, the systematic use of this keyword is the only way to obtain sound and reusable MI semantics in the general case—called by Sakkinen [1989; 1992] *fork-join inheritance*. In contrast, when the `virtual` keyword is omitted, one obtains what we shall term *non-virtual multiple inheritance* (NVI), which provides a cheaper implementation but degraded semantics (Appendix A.1). Therefore, in this section, we would have to use `virtual` to annotate every superclass. These precautions are not necessary for a language of sounder constitution like Eiffel.

The dissymmetry in the implementation between attributes and methods is consistent with the dissymmetry between attributes and methods in the C++ language specifications, especially with multiple inheritance. Consider the diamond of Figure 3 and suppose that an attribute `foo` is introduced in both B and C. This will yield, in D, two distinct attributes with the same name `foo`. This is consistent with the programmer's intuition—interested readers are referred to [Ducournau and Privat 2008] for more formal arguments. Suppose now that the programmer decides to define a reader method, `get_foo`, in both B and C. Then there will be only a single reader in D. However, the dissymmetry in the specification is not a consequence of the dissymmetry in the implementation—for instance, Ducournau and Privat [2008] propose solutions to multiple inheritance conflicts that could work with subobjects for distinguishing between the two methods `get_foo` in D.

3.2 Casting

Subobjects make casting real. It is now a matter of *pointer adjustment*. So casting implementation involves encoding Δ's in the object representation. Δ's represent 1-dimensional relative positions, hence they verify two trivial properties:

$$\forall \tau_d, T, U, V : \qquad \tau_d \preceq T, U, V \Rightarrow \Delta_{T,V} = \Delta_{T,U} + \Delta_{U,V} \tag{3}$$

$$\forall \tau_d, T : \qquad \tau_d \preceq T \Rightarrow \Delta_{T,T} = 0 \tag{4}$$

3.2.1 *Upcast.* Changing from the current subobject, with type τ_s, to a statically known supertype T, requires a shift $\Delta_{\tau_s,T}$, which depends on τ_d. An extra table, noted $\Delta^{\uparrow}_{\tau_s}$, is needed in the method table of each subobject. T offsets in $\Delta^{\uparrow}_{\tau_s}$ are ruled by an invariant similar to method Invariant 3.3, page 18:

Invariant 3.4 (Type-dependent upcast). *Each class T has an offset in the static context of each of its subclasses, that is invariant with respect to τ_d.*

This offset is denoted $\iota_{\tau_s}(T)$, where $\tau_s \preceq T$, and it is, unlike Δ's, independent of τ_d: then $\Delta_{\tau_s,T} = \Delta^{\uparrow}_{\tau_s}[\iota_{\tau_s}(T)]$, shortened in $\Delta^{\uparrow}_{\tau_s}(T)$. Instead of being a proper table, $\Delta^{\uparrow}_{\tau_s}$ can be better inlined in the method table and the shifts are now ruled by the same Invariant 3.3 as methods. Indeed, upcasts can be understood as if every target class introduces a method for upcast towards itself. Upcast could be actual methods, but it is more efficient to put a shift in the method table entry, instead

Table III. Cast structures for the class $\tau_d = D$ from Fig. 3, and each of its static supertypes (τ_s)

τ_s	$\Delta_{\Downarrow}$	$\Delta^{\uparrow}$	$\Delta_{\downarrow}$	$\Delta^{\Uparrow}$
A	$\Delta_{A,D}$	[0]	$((A,0)(B,\Delta_{A,B})$ $(C,\Delta_{A,C})(D,\Delta_{A,D}))$	
B	$\Delta_{B,D}$	$[\Delta_{B,A},0]$	$((B,0)(D,\Delta_{B,D}))$	
C	$\Delta_{C,D}$	$[\Delta_{C,A},0]$	$((C,0)(D,\Delta_{C,D}))$	
D	0	$[\Delta_{D,A},\Delta_{D,B},\Delta_{D,C},0]$	$((D,0))$	$((A,\Delta_{D,A})(B,\Delta_{D,B})$ $(C,\Delta_{D,C})(D,0))$

$\Delta_{\Downarrow}$ is a scalar, $\Delta^{\uparrow}$ a vector, while $\Delta_{\downarrow}$ and $\Delta^{\Uparrow}$ are association structures such as hashtables. The mnemonic is as follows: single and double arrows are respectively relative to static and dynamic types; upwards and downwards arrows denote the direction of the adjustment, towards super- or sub-classes.

of an address. This is complementary to the equivalence discussed in Section 2.6, page 15, but useless in invariant-reference implementations.

3.2.2 *Cast to Dynamic Type.* As references are relative to static types, an equality test between two references first requires that their types are equal. Without loss of generality, both references must be reduced to their dynamic type—so each method table must contain the shift Δ_{τ_s,τ_d} (denoted $\Delta_{\Downarrow}^{\tau_s}$), at an invariant position. When the two types are in a subtyping relation, a single upcast is sufficient.

3.2.3 *Accessing Inherited Attributes.* Invariant 3.2 only works when the receiver is typed by the class introducing the attribute. For other accesses, an implicit upcast is required. Let $\delta(p,U)$ be the position of an attribute p with respect to the subobject of type U, and δ_p be the offset of p in the type T_p ($\tau_s \prec T_p$) that introduces it (Invariant 3.2), whereby $U \preceq T_p$. Then:

$$\delta(p,\tau_s) = \Delta_{\tau_s,T_p} + \delta(p,T_p) = \Delta^{\uparrow}_{\tau_s}(T_p) + \delta_p \tag{5}$$

Therefore, when the attribute is not introduced in the static type of the receiver ($T_p \neq \tau_s$), there is appreciable overhead, italicized below, in comparison with SST:

```
load [object + #tableOffset], table
load [table + #castOffset], delta
add object, delta, object                                   3L + 1
load [object + #attributeOffset], attribute
```

In the worst case, assignment $a.x := b.y$ needs three upcasts, for access to a and b, and between y and x types. Some parallelism is likely but the sequence is 5-fold longer than in SST, with respect to the cycle count ($5L + 3$ vs. $L + 1$) and instruction number (11 vs. 2, see Appendix D.2, page App–25).

3.2.4 *Downcast.* With subobjects, downcasts add a pointer adjustment to the dynamic subtype check. So a cast towards the static subtype T requires the shift $\Delta_{\tau_s,T}$ once the test has succeeded. Moreover, in contrast with SST, direct access to the desired data and constant-time are difficult, at least in separate compilation. Indeed, the equivalence between subtype testing and method invocation (Section 2.6, page 15) does not hold here.

Without loss of generality, each class τ_d needs an association structure which maps each supertype T to $\Delta_{\tau_d,T}$—this structure, denoted $\Delta^{\Uparrow}$, can be referenced by each method table, not only by τ_d. A downcast from τ_s to T looks for T in the

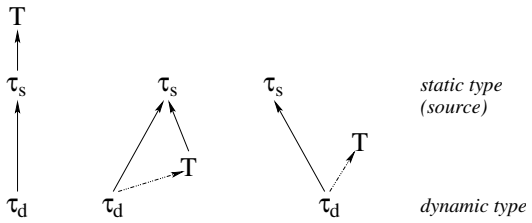


Fig. 5. Upcast (left), downcast (center) and side cast (right), to a target type T—the dashed subtype relation must be checked at run-time before shifting.

table. A failure occurs if T is not found. Otherwise, $\Delta_{\tau_d,T}$ is returned and:

$$\Delta_{\tau_s,T} = \Delta_{\tau_s,\tau_d} + \Delta_{\tau_d,T} = \Delta^{\tau_s}_{\Downarrow} + \Delta^{\Uparrow}(T) \tag{6}$$

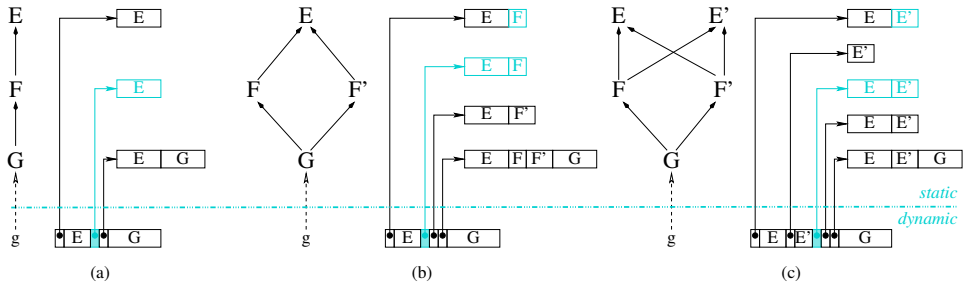
Note that the tables $\Delta^{\uparrow}_{\tau_d}$ and $\Delta^{\Uparrow}$ have the same contents, with different structures and usages—the former is used when $\tau_s = \tau_d$ and τ_s is statically known, whereas, in the latter, τ_d is statically unknown (Table III). Instead of $\Delta^{\Uparrow}$, it might be advantageous to use static types, by associating a new table $\Delta^{\tau_s}_{\downarrow}$ to each subobject, i.e. the restriction of $\Delta^{\Uparrow}$ to the classes between τ_s and τ_d: $\Delta^{\tau_s}_{\downarrow} = \Delta^{\Uparrow}/[\tau_s, \tau_d] + \Delta^{\tau_s}_{\Downarrow}$. This would avoid a two step downcast:

$$\Delta_{\tau_s,T} = \Delta^{\tau_s}_{\downarrow}(T) \tag{7}$$

but the advantage in time is small and at the expense of memory overhead. Moreover, with $\Delta^{\Uparrow}$, downcasts can be generalized to *side casts* (aka *cross casts*), where the target type T is a supertype of τ_d, but not a subtype of τ_s (Figure 5, right).

Direct Access Matrix. A truly constant-time implementation of $\Delta^{\Uparrow}$ would require coding the specialization partial order. This is a difficult problem in MI, even in the boolean case. There is, however, a technically simple solution which provides direct access—namely, a $N \times N$ matrix, where N is the class number: $mat[id_T, id_U]$ contains $\Delta_{T,U}$ if $U \preceq T$, and otherwise a distinguished value for failure. Such a matrix requires $2N^2$ bytes, i.e. $2N$ bytes per class, which is a reasonable cost when there are $N = 100$ classes, but it is not when $N >> 1000$ (see Appendix C). Class identifiers id_C must be computed globally for all classes C. A convenient injective numbering of N classes is to associate a number in $1..N$ with each class, in such a way that $id_C > id_D$ when $C \prec D$, i.e. numbering is a *linear extension* of $\prec$. This is the natural chronological numbering in class load ordering. Then the square matrix *mat* can be replaced by a triangular one which is in turn split into one vector *vect* per class, using $vect_U[id_T]$ instead of $mat[id_T, id_U]$. The cost is reduced to an average of N bytes per class. This is the worst-case cost of Cohen's display in SST (Section 2.3.2, page 12) but the average cost of the SST implementation is far lower.

Overall, these direct access tables are over space-consuming and we shall examine different practical alternatives in Section 4, especially *perfect hashing* which is, to our knowledge, the first constant-time solution to this problem with reasonable space occupation.



Each diagram depicts a single instance (below the dashed line) and its method tables (above the line). (a) for all $G \prec F$, E and F subobjects may be merged, (b) for such a G, merging of either F or F' is possible, not both, (c) same thing, however F might be merged in E, and F' in E' if the prefix condition were satisfied for both. Grey parts show the result of merging F and E.

Fig. 6. Three cases of empty subobjects, with static (a) and dynamic (b,c) bottom-up merging

3.3 Empty Subobject Optimization (ESO)

On the basis of this subobject-based implementation, a simple optimization can markedly reduce space overhead. Indeed, an exception to the type-dependent reference Invariant 3.1 is possible when a subobject is empty, i.e. when the corresponding class, say F, introduces no attributes. In this situation, a bottom-up merging of the F subobject within the subobject of some direct superclass of F, say E, can be considered. Statistics in Table V, page App–15, show that this situation is rather frequent at most of our benchmarks.

Several cases should be distinguished. In the first case (Figure 6-a), E is the only direct superclass of F, and F introduces no methods either; thus E and F have the same set of methods. The F subobject can then be merged into the E subobject as, without merging, the contents of both method tables (i.e. method addresses) would be the same. Here, merging is invariant with respect to dynamic types, i.e. E and F subobjects are merged in all subclasses of F. Multiple inheritance problems are avoided because F has no more methods than E—if another subclass F' of E is in the same situation, then the subobjects E, F and F' can be merged together in any subclass of both F and F'. In some way, this merging is a property of F—method tables are shared and shifts between F and E are avoided, as $\Delta_{F,E} = 0$ and $\iota_{\tau_s}(E) = \iota_{\tau_s}(F)$, for all $\tau_d \preceq \tau_s \preceq F \prec_{\mathrm{d}} E$. The code generated for all $\tau_s \preceq F$ takes the merging of E and F into account—e.g. access to an attribute introduced in E on a receiver typed by F will need no cast.

In the second case, F has either more methods than E (Figure 6-b), or more than one direct superclass (Figure 6-c)—the latter condition implies the former, as superclasses yield upcast pseudo-methods. If the method ordering of E is a *prefix* of the method ordering of F (i.e. the offsets of E methods are the same in F), then the E and F subobjects may be merged in the implementation of $\tau_d \preceq F$. However, E must not be already merged within another F', in the same $\tau_d \preceq F'$. Once again, merging works in this restricted case because the aforementioned *prefix condition* (Section 2.6, page 15) is also invariant with respect to dynamic types. However, E and F will not be merged in all subclasses of F. This means that merging is not a property of E, F or even G, but only of some $\tau_d \preceq F$. In this case, merging saves

space but not time. The code generated for all $\tau_s \preceq F$ cannot consider that E and F are merged (hence $\iota_{\tau_s}(E) \neq \iota_{\tau_s}(F)$), but the data structures for some $\tau_d \preceq F$ may do this merging. In particular, access to an attribute introduced in E, on a receiver typed by F, needs a cast, but the shift will generally be null ($\Delta_{E,F} = 0$). The case where the E subobject is itself empty must be carefully examined—this roughly amounts to merging F within the superclass in which E is merged.

Finally, top-down merging can also be considered—for instance, in Figure 6-b, F could be merged with E, and F' with G. F may also be a *root*. However, top-down merging cannot be static because of possible multiple inheritance in subclasses.

The prefix condition is the basis of SST Invariant 2.2. In MI, it is always partly satisfiable; indeed, as the method ordering of a class is arbitrary, it can extends the method ordering of at least one of the direct superclasses. Therefore, almost all empty subobjects can be merged—the only exception is when one attempts to merge two different subobjects into the same subobject. In this rather exceptional case, only one merging is possible.

The statistics in Table VII, page App–17, show that empty subobjects are frequent and that this simple optimization is essential for reducing the subobject overhead in the object layout. This optimization might already be known—however, if this is the case, it has not been widely noted. Some allusions in [Lippman 1996] suggest that it might be used in actual compilers—*My recommendation is not to declare nonstatic members within a virtual base class. Doing that goes a long way in taming the complexity* (p. 139). *In general, the most efficient use of a virtual base class is that of abstract virtual base class with no associated data members* (p. 101). However, after nonexhaustive experiments with several compilers, for instance, `gcc` from 2.2 to 4.3 and SUN 5.3 C++ compilers, we could not find any evidence of its use.

3.4 Evaluation

The overhead of this multiple inheritance implementation is marked and touches all considered aspects. The main drawback is that the overhead is the same when MI is not used. Indeed, separate compilation is unable to detect that a given class is always specialized in SI.

Dynamic Memory. In each object, the overhead is equal to the number of indirect superclasses of the object class. Statistics in Table VII, page App–17, show that this number could even be greater than the attribute number. With ESO, it could be restricted to superclasses which introduce at least one attribute.

Static Memory. The number of method tables is no longer the class number N but the cardinality of the inheritance relationship $\preceq$. Thus it is quadratic in N in the worst-case, whereas the total size $\sum_C(\sum_{C \preceq D} M_D)$ is no longer quadratic, but cubic. This cubic worst-case is a property of subobject-based implementations and an inescapable consequence of multiple inheritance—static pointer adjustments require cubic-size $\Delta^{\uparrow}$ tables. However, the formula should be corrected to take ESO into account: $\sum_D$ is restricted for D's which are not merged within some E, with $C \preceq D \prec_{\mathrm{d}} E$. Statistics confirm both the overhead of this implementation and the benefits of empty subobjects. In a mainly *non-virtual* inheritance context (Appendix A.1), Driesen [2001] reports a ratio larger than 3 on the table sizes, vs.

the hypothetical SST size ($\sum_C M_C$). In a pure virtual inheritance context, our statistics show that the ratio may exceed 6, but that it is reduced to 4 with ESO (Table XII, page App–21). When taking shifts into account, the ratio climbs again to 6. A priori, *thunks* seem to be roughly equivalent to putting shifts in method tables: (i) there are less *thunks* than table entries; (ii) the code sequence is two instructions shorter; but (iii) they need two extra words instead of one. However, statistics show that thunks are actually more costly for large hierarchies (Tables XIV, page App–22, to XV).

Time Efficiency. Besides initialization, instance creation is now linear in the superclass number, since a pointer to method table must be assigned for each subobject. A shift is needed each time an assignment or a parameter passing is polymorphic. This imposes extra access to the method table. The real impact on method calls is more questionable, i.e. shifting might be done within the processor's latencies or in parallel. Experiments by [Driesen 2001] seem to give a small advantage to thunks over shifts. But this conclusion is based on classic C++ programs making heavy usage of *non-virtual* inheritance—in this situation, shifts are mostly null and the thunk is the method itself (Appendix A.1). Although no conclusion about full MI—i.e. pure *virtual* inheritance—can be drawn, it seems that shifts have a marked overhead compared to SST implementation. This is not surprising since, even though the cycle counts are almost equal, this is at the expense of some parallelism, and it is likely that shifts take the place of some other code sequence. Furthermore, regarding downcast, some actual implementations seem quite inefficient, with an explicit function call and a cost that is apparently linear in the number of superclasses [Privat and Ducournau 2005]. This could be improved and we describe an inlined constant-time test using perfect hashing in Section 4.2.

Monomorphic Cases. Intra-procedural analysis allows the compiler to apply the same optimizations as in SST. Besides static calls and `null` tests, monomorphism may also optimize upcasts as all shifts are known at compile-time, as is τ_d. Therefore, there is almost no MI overhead in the monomorphic cases. In the case where the call is static but the receiver is still polymorphic—e.g. when the callee has been declared `final`—this advantage vanishes, as an adjustment may be required.

Runtime Production. Like the SST implementation, this MI implementation is fully compatible with separate compilation and dynamic loading. All offsets might be computed at link/load time, even though this is not the case in actual C++ compilers. A JAVA-like virtual machine could actually be specified in full MI with this implementation, but the addition of overheads, induced by MI and virtual machine, would likely weaken the resulting efficiency.

Abstraction Cost. In contrast with SST implementations, splitting a class (Figure 2, page 14) will cause some dynamic overhead—adding a subobject increases the object size together with the need for pointer adjustment. This is a severe counter-argument to the central OO philosophy of reusability. ESO would partly counter-balance this overhead, as splitting a class increases the ESO probability.

Conclusion on Subobjects. Although C++ is the only example of the implementation presented in this section, we shall call it *standard subobject* MI implementa-

tion (SMI), as it appears to be the single known implementation that is compatible with (i) sound MI semantics, (ii) separate compilation and dynamic loading, while being (iii) time-constant and (iv) space-reasonable. Moreover, the presented variant is likely the most commonly implemented in actual C++ compilers. The complexity of this "standard" explains part of the reluctance to use multiple inheritance as well as some unsound C++ features. Appendix A examines small variations in order to convince readers that this "standard" is a good choice.

4. ALTERNATIVES TO MULTIPLE INHERITANCE UNDER THE OWA

This section examines a few alternatives that do not rely on subobjects, but rather on the SST reference Invariant 2.1, while remaining compatible with dynamic loading. First, attribute implementation can be reduced to methods, through *accessor simulation* (Section 4.1). Then, general approaches like hashing (Section 4.2) and caching (Section 4.3) are considered.

4.1 Attribute Accessors and Accessor Simulation

As already mentioned in Section 2.6, page 15, attributes and methods are similar entities that can be conceptually handled in similar ways. Without loss of generality, attribute implementation can be encapsulated in accessor methods—when an attribute position differs between a class and its direct superclass, one needs only to override its accessors. In that way, attribute offsets do not matter.

True accessors require a method call for each access, which would be inefficient. However, a class can just simulate accessors by replacing in the method table the method address with the attribute offset. This is called *field dispatching* by Zibin and Gil [2003b]. Finally, attributes can be grouped by introduction class, with an invariant position in the group similar to Invariant 3.2. One can then substitute, for their different offsets, the single relative position of the attribute group, stored as a method in the method table. This is *accessor simulation* (AS). If the underlying implementation is that of SST, the access code is the same as with standard MI implementation in the general case ($T_p \neq \tau_s$, page 21).

Thus, accessor simulation represents a kind of subobject-based implementation, with a single method table but ubiquitous shifts for all attribute accesses, even when $\tau_s = T_p$. However, all other pointer adjustments are avoided. In order to save shifts in the frequent case of access to `self`, Myers [1995] proposes a double compilation of each class. In the first one, `self` attributes are compiled with shifts, whereas in the second one, they are compiled without shifts, under the assumption that the group position will be preserved in all subclasses (e.g. if the class is only specialized in SI). The appropriate version is chosen at link-time (Section 6.5, page 43).

Accessor simulation can be used with any implementation of method invocation. Static typing is also required, because an attribute must be introduced by a single class—however, SMALLTALK like encapsulation can replace static typing (Section 6.2.1, page 37). Nevertheless, accessors are a solution for attributes only when the question of methods has been solved. If the object layout may consist of a table where attribute offsets do not matter, efficient access to the introduction class is needed. Thus accessors are not an alternative in themselves but we shall see different ways of incorporating accessor simulation in a general implementation framework which preserves object reference variance (Section 5.1.3, page 32) or re-

covers SST invariance with a global computation (Section 6.3, page 39), hashtables (Section 4.2) or a flow of method tables (Appendix A.4).

4.2 Hashing and Perfect Hashing

The three mechanisms that we investigate are a matter of searching for an attribute value, a method address or a class ID. A general approach to searching consists of hash structures, that would associate, in each class, some key (the method, attribute or class ID) with the desired data. Here, hashtables can be used in several ways:

—With subobjects (SMI), $\Delta^{\Uparrow}$ tables need an implementation for subtype testing (Section 3.2.4, page 21); the hashtable associates a shift with a class ID and is pointed to by the method table of each static type.

—With invariant reference, method invocation could be implemented with a hashtable associating a method address with a method ID. However, methods are quite numerous, and the space-occupation ratio of hashtables is not very good. Hence, a better solution relies on the equivalence between method invocation and subtype testing (Section 2.6, page 15), and the class ID is now associated with the position of the group of methods introduced by the considered class. No specific data is required for subtype testing since the class ID suffices. Finally, the hashtable can be inlined at negative offsets in the method table, making it *bidirectional* (Section 6.3.3, page 41).

—Using hashtables for the object layout would be quite inefficient, but accessor simulation can be used instead (Section 4.1, page 26)—therefore, the class ID would be associated with both offsets of the method and attribute groups.

This is thus the general way of using hashing for the implementation of the three considered mechanisms, which all amount to hashing class ID's. The point is, now, to determine the precise hash structure that would provide an efficient solution. A simple hashtable implementation is *linear probing* [Knuth 1973; Vitter and Flajolet 1990]. A hashtable is then a simple array of size H, coupled with a hashing function $h : \mathbf{N} \to [0..H-1]$. The key k is hashed and searched in the array, starting from offset $h(k)$, until k or an empty entry is found. At the array end, the search continues from offset 0. The hashtable efficiency is still a space/time tradeoff and linear probing is considered very efficient when the occupation ratio is about 2. As each class must hash the ID's of all of its superclasses, the number of occupied entries will range from 0 to several tens. A uniform parameter H would make the hashtables over space-consuming, since H must be greater than the number of hashed keys, i.e. superclasses. Therefore, H must depend on the considered class. We now consider a class C, with its hashtable of size H_C coupled with a hashing function h_C that is parametrized by H_C. Hence $h_C(x) = hash(x, H_C)$, where *hash* is some hashing function that must be very simple, for instance the remainder of integer division (denoted `mod`), or bit-wise `and`.

The considered hashtables present a unique property, as they are *immutable*—i.e. once they are computed, at link or load time, there is no longer any need for insertion or deletion. Therefore, for each class C, knowing the identifiers of all of its superclasses, it is possible to optimize the hashtable in order to minimize the average number of probes and the table size H_C. In the ideal case, H_C may

be defined as the least integer such that all tests need only one probe, i.e. h_C is injective on its set of identifiers. This is known as the *perfect hashing* problem [Sprugnoli 1977; Czech et al. 1997]. Such an approach was proposed by Ducournau [2008] for subtype testing in the general case, and method invocation in the special case of JAVA interfaces (Section 5, page 30). The first abstract evaluations show a time/space tradeoff between the two considered hashing functions; H_C parameters are markedly lower with modulus, but integer division has a high latency whereas `and` takes one cycle. However, an improved technique, called *perfect class numbering*, provides more compact tables by optimizing class identifiers [Ducournau and Morandat 2009].

This use of perfect hashing is the only constant-time, linear-space, purely incremental technique that we know for subtype testing in multiple inheritance. It also works for method invocation and recent experiments in the PRM compiler show that the technique is quite efficient for both mechanisms [Ducournau et al. 2009], hence it could be used for implementing JAVA interfaces. However, its application to full multiple inheritance requires further study.

4.3 Caching

When the searching efficiency is not satisfactory, caching (aka memoization) is a usual way of improving it. In the object-oriented implementation context, the technique has been mostly used with dynamic typing, that is in a context where the known techniques are rather inefficient (at least under the OWA). It also applies, of course, with static typing, but we must be cautious because of the higher efficiency of the underlying implementations. The cache can be attached to a variety of data structures: (i) the code, for each call site; (ii) the receiver, in its method table; (iii) the method itself, as a global data structure. In all cases, the implementation must rely on some general method invocation technique, for instance perfect hashing (Section 4.2).

4.3.1 *Type Prediction and Inline Cache.* Type prediction stems from the idea that a given type may be considered as more likely for some method call sites, for instance type `integer` for the method `+`. Hence, type prediction consists of compiling a method call by a comparison between the expected type and the actual receiver type; if the test succeeds, a static call is done, otherwise the call must use the underlying technique. The type prediction efficiency closely relies on the branching prediction of modern processors—a well-predicted conditional branching saves on B cycles. If the expected type is well chosen and the underlying implementation is not that efficient, type prediction can be a marked improvement.

There are many variations around this basic idea. The expected type may be that of the receiver for the previous call on the same call site (*inline cache*). Prediction may also be *polymorphic* when the receiver type is tested against several types [Hölzle et al. 1991]. In this case, the call sequence is a small *decision tree* which compares the type of receiver with all expected types.

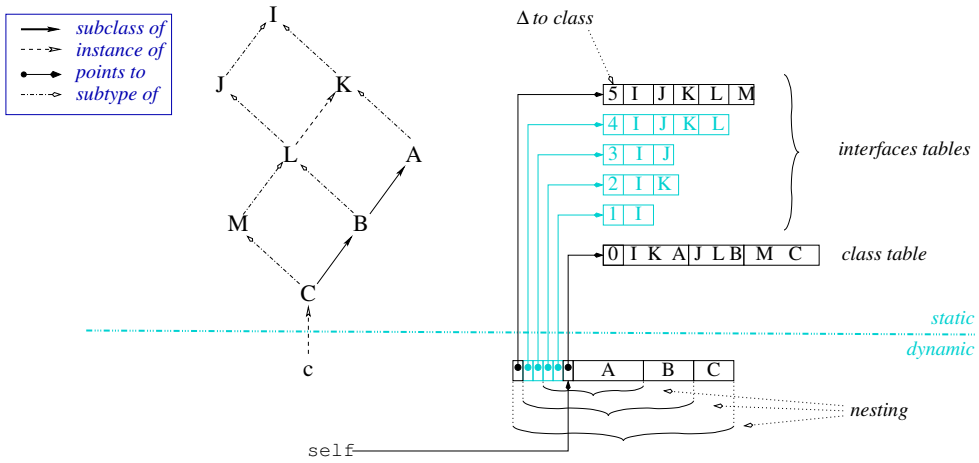
The technique can apply to all three mechanisms. For subtype testing, the cache only needs the type ID. For method call and attribute access, the cache must also memoize, respectively, the method address and the attribute position.

4.3.2 *Cache in Method Tables.* When the cache is in the method table, the solution is similar. However, instead of caching the method ID, it is preferable to cache the introduction class ID. This increases cache hits. The method address must thus be replaced by the address of the group of methods introduced by the considered class. Attribute access would use accessor simulation, with the same way of caching. When the technique is used for several mechanisms, the cache may be common to all concerned mechanisms, or specific to each of them. Caching can be efficient only when the cache-miss rate is low enough. Palacz and Vitek [2003] report an experimentation of subtype testing that shows that cache-miss rates can be as low as 0.1% or more than 50%, according to the different benchmarks. In the former case, caching can improve many techniques when they are not too efficient; indeed, the cache-hit case is exactly equivalent to Cohen's display for subtype testing, and hardly better than bit-wise-`and` perfect hashing for method invocation. In contrast, when the cache-miss rate is 50%, caching should be an improvement only for very inefficient techniques, and searching for more efficient techniques is likely better. Cache-hit rates can be further improved with multiple caches. With n caches, the cache data structure is replicated n times in the method tables, and classes are statically partitioned into n sets, for instance by hashing their name (at compile-time) or their ID (at load-time). Finally the code of each call site uses the cache structure corresponding to the given site. Hence the cache-miss rate should asymptotically tend to 0 as n increases. In this approach, the tables of the underlying implementation are only required to contain class or interface ID's for which there is a collision on their proper cache.

Ducournau et al. [2009] tested this approach in conjunction with perfect hashing, in the PRM testbed (Section 6.5.2, page 44). A first observation is that specific caches improve the overall cache-miss rate which is about 20% with a quadruple cache. However, caching improves perfect hashing only with modulus hashing function, on processors where integer division is quite inefficient. Caching is often used in the context of JAVA interfaces (Section 5.3) and JIT compilers (Section 6.6). The efficiency observed in this context is likely due to the fact that the number of cached entities is rather low—only interfaces that cannot be further optimized by JIT optimizations—hence dramatically reducing the cache-miss rate.

4.4 Conclusions on Alternative Implementations

Perfect hashing (PH) and accessor simulation (AS) seem very appealing, though in a limited setting. Perfect hashing should be considered for JAVA interfaces (Section 5, page 30) and for general subtype testing. Accessor simulation might complement an implementation that does not address attribute access, like perfect hashing. However, it yields marked overhead and it does not seem that this combination (i.e. PH and AS) could provide an efficient alternative to subobjects. Type prediction gives rise to *binary tree dispatch* that is used in a global compilation setting (Section 6.2.2, page 38). It could also find specific niches, for instance for implementing *tagging* of primitives types in order to avoid automatic *boxing* (see Appendix B.1). In contrast, method table caching is likely not an improvement when the underlying implementation is at least as efficient as bit-wise-`and` perfect hashing, unless cache-misses are exceptional. However, exceptional cache misses cannot be considered to be a scalable assumption.



3 classes A, B, C and 5 interfaces, I, J, K, L, M. I stands for `Object`. In grey, shared interface tables.

Fig. 7. Single inheritance and multiple subtyping: multiple inheritance variant

5. SINGLE INHERITANCE AND MULTIPLE SUBTYPING (MST)

Between the two extreme cases of SST and MI is the middle case where classes are in SI but types are in multiple subtyping (MST), whilst class specialization remains a special case of subtyping. JAVA is a typical example [Gosling et al. 2005], whereby the `extends` relation between classes is single, and the `implements` relation between classes and interfaces and the `extends` relation between interfaces are multiple. Interfaces are *abstract* classes and only define method signatures—static variables are not considered here since they are not object-oriented. Many languages have a very close type policy, for instance THETA [Myers 1995] and all languages designed for Microsoft .NET like C# [Microsoft 2001]. Furthermore, the absence of multiple subtyping was viewed as a deficiency of the ADA 95 revision, and this feature was incorporated in the next version [Taft et al. 2006]. This middle case deserves examination especially in the framework of separate compilation and dynamic loading that also represent a key requirement of JAVA and .NET platforms. In such a framework, the SST Invariants 2.1 and 2.2 cannot hold together for interfaces—that is, with the natural SST class implementation, a method introduced by some interface will be located at different offsets in the different classes which implement the interface. However, standard MI would be too complicated, as both invariants could hold for classes. Hence, two different solutions can be designed, by simplifying SMI implementation (Section 3, page 17) or complicating SST implementation (Section 2, page 9).

5.1 Multiple Inheritance Variant

Standard MI implementation can be notably simplified in the present case. Of course, empty-subobject optimization (ESO, Section 3.3, page 23) would apply to all interface subobjects. However, as interfaces are expected to introduce some methods, only the second merging case works. A more specific approach is better.

5.1.1 *Principle.* Starting from standard MI implementation, the first step consists of conciliating different method tables with attribute invariance. There is thus a single subobject for the class and all its superclasses, with a single method table for this subobject—both are organized as in SST. Besides this single class subobject, all SMI invariants hold. All interface subobjects are empty. Therefore, pointers to method tables can be grouped in a header that makes object layout *bidirectional*—positive offsets are for attributes, negative offsets are for interface tables, and offset 0 points to the class table. Each interface table begins with the offset of the pointer to this table, which stands for $\Delta^{\tau_s}_{\Downarrow}$, and this value will be used for shifting the receiver in a method call when it is typed by an interface. No shift is needed for a class-typed receiver, as it points to offset 0, whereas an interface-typed reference points to the offset corresponding to the interface.

In a second step, the header is ordered in such a way that the superclass implementation is nested inside the subclass one—the subclass adds new interfaces at negative offsets and new attributes at positive offsets (Figure 7).

INVARIANT 5.1. *Superclass implementation is nested inside subclass implementation, so interface shifts ($\Delta^{\tau_s}_{\Downarrow}$) are invariant with respect to dynamic types.*

The third step involves factorizing interface tables. Table sharing relies on the aforementioned *prefix condition* (see Section 2.6, page 15). Each interface or class orders its super-interfaces in some arbitrary top-down order such that the superclass order is always a *prefix* of the subclass order and the interface order is a prefix of that of at least one of its superinterfaces. Methods are also grouped in all tables according to this order, and method offsets are invariant in each group, as with Invariant 3.2. Two interfaces can share the same table when the superinterface order is a prefix of the subinterface order. In Figure 7, the class table is shared by K and the table of M is shared by all other interfaces.

The code for class access is the same as in SST for all three mechanisms. When the receiver is typed by an interface, the code for method invocation is the same as in MI (page 19), but `#deltaOffset` does not depend on the method. Finally, instead of being true method tables, the interface tables may contain only the addresses of the method groups in the class table. This will save space, at the price of an extra `load` that cannot run in parallel contrary to that of SMI. However, this would hinder sharing between interface and class tables.

Overall, sharing may be quite effective in practice, but it does not seem that it reduces the cubic worst case because an interface can verify the prefix condition only with a single direct super-interface.

5.1.2 *Casting.* Several cases must be considered: (i) from class to class, this is done exactly as with SST (Section 2.3.2, page 12); (ii) from interface to interface, downcasts are as with SMI (Section 3.2.4, page 21); (iii) from interface to interface, upcasts are only required when both interfaces do not verify the prefix condition, e.g. from L to K in Figure 7; (iv) for an upcast from class to interface, the shift is static (constant and invariant with respect to dynamic type, thus without table access) thanks to nesting; (v) a sidecast to interface is like in SMI; (vi) from interface to class, the shift is in the table header, and the test required by downcast is done as in SST. When a downcast reduces to SMI, $\Delta^{\Uparrow}$ tables are needed.

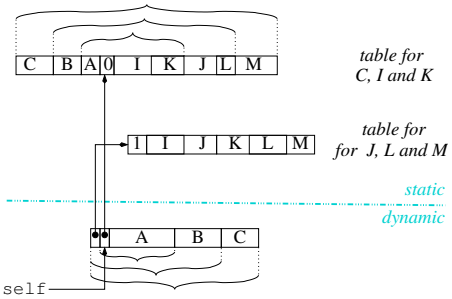


Fig. 8. THETA techniques, for class C from Figure 7

5.1.3 *The* THETA *Approach.* Some optimizations to this technique have been proposed for the THETA language [Myers 1995]. The basic idea is to extend object-layout bidirectionality to method tables. The positive part contains methods introduced in interfaces when the negative part contains methods introduced in a class (Figure 8). In the example, sharing is not better than with the first variant (Figure 7), but it is intuitive that bidirectionality might improve sharing. Indeed, without bidirectionality, methods introduced by a class (say A) prohibit any sharing for the subclass interfaces (J and L). In Figure 7, if the interface order of L were $IKJL$ instead of $IJKL$, a third table would be required for IJ. However, with the bidirectional approach, the $IKJLM$ table could be shared with the class table. An optimizing algorithm for computing tables is also proposed.

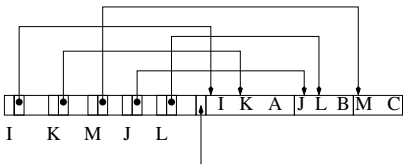
5.2 Single Subtyping Variant

The alternative involves extending the SST implementation to interfaces.

5.2.1 *Principle.* Reference Invariant 2.1 is now maintained in all cases, whereas position Invariant 2.2 is still restricted to class-typed entities. Object layout and class method tables are the same as with SST, but some data must be added in method tables to deal with interface-typed entities. A data structure is needed to find, from each interface implemented by a class, the corresponding methods, and for downcast. Once again, it is more efficient to group methods introduced by the interface in such a way that the data structure associates the group address (or offset) with the interface ID. Finally, the data structure can be inlined at negative offsets in the method table, making it *bidirectional* and saving one indirection.

Direct Access Matrix. Constant-time direct access to these data structures for a given interface would imply non-conflicting numbering of interfaces. With dynamic loading, an injective numbering is a solution which may yield large but mostly empty tables, as interfaces may be quite numerous (up to 1000 in the largest benchmarks, see Appendix C). The simulation carried out in [Ducournau 2008] clearly shows that this would be over space-consuming on large hierarchies.

Association Structure. Without loss of generality, the solution will consist in an association structure such as a hashtable, similar to $\Delta^{\Uparrow}$, within the class method table (Section 3.2.4, page 21). *Perfect hashing* (Section 4.2, page 27) offers constant-time access, with $4L + B + 3$ cycle count and roughly linear-size tables. Figure 9



The method table is bidirectional. Positive offsets represent the method table itself, as in Fig. 7. Negative offsets represent the association structure, e.g. a hashtable, which maps each interface ID to a method group address.

Fig. 9. Multiple subtyping—method table in single subtyping variant, for class C from Fig. 7.

depicts the implementation of the class C from Figure 7. Interested readers are referred to [Ducournau 2008; Ducournau and Morandat 2009] for detailed implementation and statistics. An alternative to hashtables might be an association list. The implied nesting of the superclass in the subclass allows the subclass association list to share that of its direct superclass—hence, this list sharing provides a space-efficient implementation that is unfortunately time-linear.

Subtype Testing. Casting is done mostly as in SST. Due to reference invariance, only downcast and dynamic type check must be considered. When the target type is a class, any SST technique applies, but an incremental one, e.g. Cohen's display is preferred. When the target is an interface, the interface association structure can be used in a boolean way.

5.2.2 *Alternative with Method Table Flow.* Implementation of interface-typed entities may provide a good opportunity for method table flow (Appendix A.4). Indeed, the space overhead of this technique would only concern interfaces.

5.3 Actual Java and .Net Implementations

According to their specifications, Java and .Net languages could adopt either variant. However, their implementation is constrained by the specifications of their run-time environment—the *Java Virtual Machine* (Jvm) [Meyer and Downing 1997] and *Common Language Runtime* (Clr) [Burton 2002]—and it would seem that all existing implementations are reference-invariant. Anyway, we cannot affirm that the specifications of Jvm or Clr enforce it. The question of JIT compilers will be discussed in Section 6.6, page 45.

Strangely enough, though method invocation and subtype testing are a matter for common implementation, we could not find in the literature any paper that addresses both mechanisms. Many of the proposed techniques apply to one mechanism but cannot generalize to the other. It is however likely that there are actual implementations where both mechanisms rely on a common implementation.

5.3.1 *Interface-Typed Receiver.* In the Jvm, a specific operation, `invokeinterface`, addresses the case of method call to an interface-typed receiver. [Alpern et al. 2001a] presents a state of the art of existing implementations, which mostly use the techniques that are described later for dynamic typing (Section 6.2, page 37). Various proposals use a large direct-access class-method dispatch matrix, possibly transformed in a smaller class-interface matrix. This is the case of virtual machines Cacao [Krall and Grafl 1997] and Sable [Gagnon and Hendren 2001]. The latter

presents an original feature—as the used class-method matrix is empty at about 95%, the empty entries are reused for allocating data. This is an interesting idea, but we have no information on how much space is saved in this way. Moreover, this approach cannot be used for subtype testing, since this mechanism requires some encoding of empty entries. Other techniques like inline cache, possibly polymorphic [Hölzle et al. 1991], are quoted. In Jalapeño [Alpern et al. 2001a], each class is associated with a fixed-size hashtable where each entry contains a method address or a compiled decision tree indexed on method ID's.

The JAVA specifications allow a method to be introduced in several interfaces—see introduction overloading, Section 2.1, page 11. Hence, with an interface-offset association, a method that is introduced by several interfaces must be replicated in the method table, with an entry in the method group of each introduction. This may slightly increase the table size. When the association structure is a shared list, the interface-offset association can be replaced by an interface-array structure where the array maps each introduced method to its offset in the method table. This, however, adds an extra indirection for a quite hypothetical space gain.

5.3.2 *Subtyping Checks.* Cohen's test is commonly used in the implementation of JAVA virtual machines for class subtyping tests [Alpern et al. 2001b; Click and Rose 2002]. However, it is often used in a non-optimal way, with separate uniform-size tables. Regarding interface subtyping tests, Alpern et al. [2001b] use a direct access array indexed by interface identifiers. According to [Palacz and Vitek 2003], the Sun Microsystems Research VM uses a linear search that is just improved with a cache in the method table. Click and Rose [2002] report a similar approach. Palacz and Vitek [2003] propose an incremental use of two non-incremental techniques, Schubert's numbering (Section 2.3.2, page 12) and coloring (Section 6.3, page 39).

5.4 Application to Multiple Inheritance and *Mixins*

The MST principle can be slightly generalized to class hierarchies that are partitioned into two kinds of classes—*primary* classes have all of the prerogatives of usual classes, but they are in single inheritance, whereas *secondary* classes are restricted in several ways. This abstract presentation covers both the so-called *mixin* approach and a proposal by Myers [1995] that extends the THETA implementation (Section 5.1.3, page 32) to full multiple inheritance.

5.4.1 *Principle of Mixins.* The literature on *mixins* (aka *traits*) is rather large [Stefik and Bobrow 1986; Bracha and Cook 1990; Ancona et al. 2003; Ernst 2002], and the usages of the term vary from formal definitions to quite informal ones. Mixins involve a variation on the notion of *abstract classes*, in a less abstract way than the interface notion of JAVA. They are often presented as a way to either avoid MI or discipline its usage. Here, our definition strictly follows the SCALA programming language [Odersky et al. 2008]. Classes are in single inheritance and a class can `extend` a single superclass `with` any number of mixins. A mixin can extend other mixins and at most a single direct superclass. An additional constraint enforces class single inheritance—when a class C extends class B with mixin M, then superclasses of M must be superclasses of B. This means that removing mixins from the transitive closure of specialization must yield SI, like removing interfaces, but the constraint on the mixin hierarchy is slightly weaker than for

interfaces. Moreover, mixins can also define attribute and method bodies, but they are abstract as they cannot create instances.

5.4.2 *The* Scala *Approach.* In Scala, each mixin M is compiled into an interface I_M and an abstract class C_M. Methods defined in M are copied into static methods of C_M that are defined with an extra parameter for the receiver. When a class C specializes M in Scala, the corresponding C class in Java implements I_M, and all methods declared in I_M are defined in C by just calling the corresponding static methods of C_M. Regarding attributes, those defined in M are defined in the C Java class, together with their accessors that are declared in I_M. Accessing such an attribute on class-typed objects is like usual Java. In contrast, when the access is typed by a mixin, e.g. in the methods of M, the Scala to Java compiler replaces the attribute access by an accessor call.

Myers proposes a similar implementation where the methods of M are copied into C. This resembles a limited form of *customization* (Section 6.4.4, page 42) that applies even in separate compilation—*mixins* are then treated like C++ *templates*, i.e. they are not compiled until they are instantiated. Indeed, the most formal presentation of mixins involves *parametrized heir classes*, i.e. classes parametrized by the superclass of the class resulting from their instantiation. Let A be a class, M a mixin, then defining a subclass B of both A and M is the same as defining B as a subclass of $M\langle A\rangle$ [Bracha and Cook 1990; Ancona et al. 2003]. This is easy to do it with the *heterogeneous* implementation of C++ templates. [VanHilst and Notkin 1996; Smaragdakis and Batory 2002], but the *homogeneous* implementation of Java makes it impossible. See also the implementations of generics in Appendix B.2.

5.4.3 *Accessor Simulation and MST Implementations.* The invariant-reference implementation presented in Section 5.2.1 and Figure 9, page 33, can be extended to a separate compilation of mixins in the following way. Attributes introduced by the mixin are grouped in the object layout, in the same way as methods in the method table. Position Invariant still holds for attributes accessed on class-typed objects. When these attributes are accessed on a mixin-typed object, accessor simulation is used. So the attribute group position is implemented as an extra method in the method table, or equivalently, the entries of the association structure can be 3-fold—mixin ID, offsets of method and attribute groups. Finally, the methods defined in the mixin are compiled as ordinary methods, apart from the fact that `self` is mixin-typed—hence, all `self`-calls in the method must use an equivalent of `invokeinterface`.

This would also apply to the multiple inheritance variant (Section 5.1, page 30), with just adding a `self`-adjustement to all mixin-typed method invocations, and thunks would likely be the right way to do it.

5.4.4 *Mixins vs. Full Multiple Inheritance.* Mixins provide an implementation advantage because they are statically identified—they are secondary classes. So the compiler can decide to compile a `self`-call as a class-typed or a mixin-typed call. When attempting to apply this approach to full multiple inheritance, the static distinction vanishes. So the compiler cannot decide between class- and mixin-typed calls. A solution similar to Myers' proposal involves double compilation—all `self`-calls will be compiled by a class- or mixin-typed calls, according to the version.

Accordingly, all other calls should be compiled as mixin-typed calls, i.e. in a less efficient way. Finally, the appropriate version would be chosen at link-time on the basis of some global analysis (Section 6.5, page 43).

5.5 Evaluation

As far as one can extrapolate actual implementations from the techniques proposed in the literature, it would seem that most JAVA and .NET implementations suffer from their interface implementation. All approaches that we are aware of present several flaws including: (i) non-constant time, (ii) non-scalable space, (iii) non-incremental, and (iv) non-applicable to both mechanisms. This could be considered bearable as long as interfaces are not intensively used. However, the JAVA API encourages intensive use of interfaces. Moreover, compiler-made programs yield numerous interfaces, for instance when they are automatically computed [Huchard and Leblanc 2000] or with SCALA, whereby mixins are transformed into interfaces. This apparent inefficiency is actually hidden by optimizations that are carried out by JIT compilers. Most interface accesses are actually compiled as class accesses, or even as monomorphic calls (Section 6.6, page 45). However, the worst-case behaviour would certainly benefit from a more efficient underlying implementation.

Two main variants must be considered. Both reduce to SST implementation when no interface is used and time efficiency is the same as SST as long as class-typed entities are concerned. With reference-invariant implementations, perfect hashing likely represents the best currently known solution, i.e. the only one that appears truly scalable and applies to both method invocation and subtyping tests. The static space overhead is rather small and the dynamic space overhead is null. The subobject-based variant could be an alternative. However, it does not address subtype testing. Thus a complementary technique, e.g. perfect hashing, must be used and the overall implementation must pay for both.

Among the various applications to full MI, mixins represent an important semantic restriction that is probably not justified by the gain in efficiency. Actually, it would also be interesting to check whether this gain is effective. It is worth noting that the reference-invariant implementation prohibits only attribute definitions, and the specification of interfaces could be extended to the definition of methods.

6. GLOBAL TECHNIQUES AND OPTIMIZATIONS

Previous sections considered only separate compilation and dynamic loading—i.e. fully incremental implementations. Separate compilation is a good answer to the modularity requirements of software engineering; it provides speed of compilation and recompilation, together with locality of errors, and protects source code from both infringement and hazardous modifications. With separate compilation, the code generated for a program unit, here a class, is correct for all correct future uses. Separate compilation thus provides the best framework for reusability which strongly suggests the *open-world assumption* (OWA). In contrast, global compilation supposes the *closed-world assumption* (CWA). In return, the additional constraints brought by the CWA give rise to new opportunities for the compiler for optimizing the generated code. Moreover, the world closure can be gradual. For instance, global linking may be envisaged as a tradeoff between global compilation and dynamic loading. Alternatively, dynamic loading can rely on temporary CWA.

In this section, we successively examine the general advantages that can be drawn from the closed world; common implementations in dynamic typing; the coloring heuristics that extend the SST implementation to multiple inheritance; various global optimizations; and their application at link- or load-time. It is essential here to distinguish between *implementation* and *optimization*; in specific cases, the latter allows the compiler to use a short cut for the former.

6.1 Advantages of the Closed World

6.1.1 *Closed Hierarchy.* When processing a class hierarchy, the first advantage of a closed world is that this hierarchy is closed—no extra class can be added unless some part of the current hierarchy is re-processed. It is then possible to know, at that time, whether a class is specialized in single or multiple inheritance, whether two $\prec$-unrelated classes have a common subclass or not, and so on. Moreover, the *external schema* (Section 1.3, page 6) of each class is known; it provides information on classes for which methods are defined. For instance, methods with a single definition may then be treated as static (monomorphic) calls. In dynamic typing, this is known as the *unique name* heuristics [Calder and Grunwald 1994]; despite its simplicity, its effect is not small as this applies to almost 45% of methods in SMALLTALK. More generally, *class hierarchy analysis* is a promising approach for optimizing programs whenever the whole hierarchy is available. "*Class hierarchy analysis*" is a common term that denotes any analysis of the class hierarchy, i.e. that relies only on the class schemata. It is also the label (CHA) of the specific analysis proposed by [Dean et al. 1995] that improves the unique name heuristics in static typing. A call is now monomorphic as soon as the callee is not overridden in the subclasses of the receiver's static type. A hierarchy analysis can also determine an optimal mixing of virtual and non-virtual inheritance (Appendix A.1) or decide which classes are primary or secondary in the application to MI of MST implementations (Section 5.4, page 34).

6.1.2 *Knowledge of Method Code.* When the considered process is the compilation itself, a second advantage is the knowledge of the code of all methods. When compiling a method, the compiler knows how the method is used, and when compiling a method call, it also knows the code for all possible callees. Many optimizations proceed from this knowledge but they all imply an underlying general implementation technique. Finally, the program entry point itself can be known—this is the key to full *type analysis*.

6.2 Implementation in Dynamic Typing

In dynamically typed languages like SMALLTALK, SELF and CECIL, the lack of type annotations makes separate compilation quite inefficient. So many techniques have been worked out in the framework of these languages—of course they all apply to static typing as well (the converse being false).

6.2.1 *Dynamic Typing and Single Inheritance (SI).* With dynamic typing, SI is no longer a simplification, because of *introduction overloading* (see page 11). When the same method name is introduced in two $\prec$-unrelated classes, the SST position Invariant 2.2, page 10, cannot hold, at least in separate compilation. This also concerns attributes unless they are *encapsulated* as in SMALLTALK. Encapsulation

reserves attribute access for `self`. As `self` has the uncommon feature of being the only statically typed entity of the language, the SST position Invariant holds for attributes, in case of SI. Accessing non-encapsulated attributes requires true accessor methods. With MI, accessor simulation can be used on all encapsulated accesses (Section 4.1, page 26). The case of attributes is rarely covered in the literature, certainly because of encapsulation.

Overall, the JAVA type system may be understood as the minimal type system required for statically typing SMALLTALK. The interface notion is the answer to introduction overloading—an interface must be defined for each method name that is introduced in several unrelated classes. Therefore, implementation of dynamic typing is at least as difficult as that of multiple subtyping (MST).

6.2.2 *Implementation Techniques.* Apart from dynamic search in the class hierarchy (aka *lookup*), method invocation techniques can be classified into *table-based* techniques, which involve constant-time direct access tables, and *tree-based* techniques, that pool techniques derived from inline cache (Section 4.3, page 28).

Compaction of the Large Dispatch Matrix. Table-based techniques involve compacting the large matrix obtained by global and injective numbering of all classes and methods. As the class and method numbers may reach and even exceed, respectively, 1000 and 10000, this table is huge, several millions of entries, and too large to be implemented as such. However, the number of valid class-method pairs is far smaller—it is the total size of method tables in SST, $\sum_C M_C$, and represents less than 5% according to our statistics (Table X, page App–19). Two compacting techniques have been proposed: row displacement [Driesen and Hölzle 1995], after a sparse table compression technique from Tarjan and Yao [1979]; and method coloring [Dixon et al. 1989]. In both cases, the result is a table, each entry of which contains a single address corresponding to some class-method pair or is empty. However, the lack of static type checking means that a method may be called on a wrong-type receiver, which may amount to either an empty entry or a class-selector pair with a different selector. Thus a dynamic type checking is needed but it reduces to a simple equality test between the expected selector and the actual one. Static typing makes this extra test useless. Coloring is detailed in Section 6.3.

Tree-Based Techniques. Tree-based techniques stem from *type prediction* (Section 4.3, page 28), which was originally proposed in a dynamic typing setting. Type prediction is not self-sufficient since mispredictions, i.e. cache misses, must be handled, when the type of the receiver is not amongst the predicted types. Therefore some underlying technique is needed, for instance a dynamic *lookup* à la SMALLTALK. An alternative is to rule out cache misses by exhausting all possible types. This requires the CWA. In practice, this is realistic only with a type analysis, at least a class hierarchy analysis (CHA) in static typing (Section 6.4.2, page 42). The expected types are then sorted by increasing class ID, which are grouped when two consecutive ID's correspond to the same method address. This gives a list of k ID intervals such that each interval corresponds to a single address and two consecutive intervals correspond to different addresses. Finally, this is transformed in a balanced tree of $k-1$ conditional branchings. This gives *binary tree dispatch* (BTD) [Zendra et al. 1997]. Hence, memory access (L latency) is avoided and the entire

call sequence is made of conditional branchings that are statistically well predicted. A predicted branch has a 1-cycle cost, whereas an unpredicted one costs B cycles. The B latency of indirect branching in table-based approaches can thus be avoided when all conditional branchings are well predicted. In contrast, the branch number k may be large and the search is in $O(\lceil log_2(k) \rceil)$. Assuming a uniform distribution of type probabilities, the average decision cost would be $log_2(k)(B+1)/2$. However, a uniform distribution is quite unlikely; in practice, the same type repeatedly occurs at the considered call site and the misprediction rate is far lower than 1/2. Driesen and Zendra [2002] analyze the efficiency of different implementation variants—if sequences, binary trees, switch tables—according to the call site patterns, i.e. the type of receiver may be constant, random, cyclic, etc.

Variants and Mixed Techniques. Interval containment [Muthukrishnan and Muller 1996] is an optimization of the latter approach that relies on Schubert's numbering. A dispatch tree based on subtype testing has also been proposed by Queinnec [1998]. It handles only SI and uses Cohen's display. There are many ways of mixing table-based and tree-based techniques, by putting either tables in tree leaves, as Queinnec, or tree roots in table entries [Vitek and Horspool 1994]. *Type slicing* [Gil and Zibin 2007] is a generalization of Schubert's numbering and interval containment to MI. The main effect of these techniques is to save static memory with respect to standard table-based techniques—the size of static data structures may be much smaller than $\sum_C M_C$. However, this is detrimental to time efficiency, i.e. as with unbounded BTD, method call is no longer time-constant. Moreover, unlike pure BTD, table access is required.

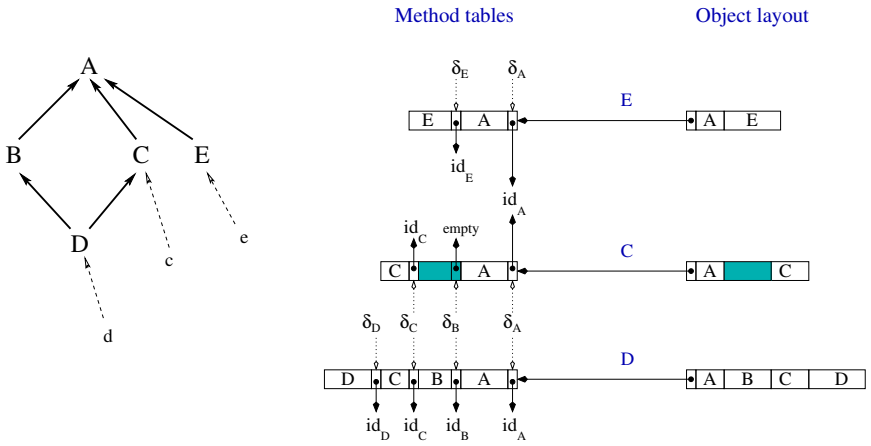
6.3 Coloring Heuristics

We now detail the coloring approach as it is quite versatile. Indeed, it applies to all three basic mechanisms; it works with dynamic typing but it is even better with static typing; and it naturally extends the SST implementation to MI without any overhead in case of single inheritance.

6.3.1 *Principle of Coloring. Coloring* can be defined as an optimization technique that retains the SST invariants and implementation at minimal spatial cost under the CWA. It has been introduced and applied, more or less independently, to method invocation (under the name of *selector coloring* [Dixon et al. 1989]), to attribute access [Pugh and Weddell 1990; Ducournau 1991] and to subtype testing (under the name of *pack encoding* [Vitek et al. 1997]). The position Invariant 2.2, page 10, is rewritten as follows:

Invariant 6.1 (Color). *Each attribute and method has an offset (color) invariant by specialization. Two attributes (resp. methods) with the same color do not belong to the same class. Each class has an offset (color). Two classes with the same color have no common subclass.*

An injective numbering of classes, attributes and methods verifies the invariant; this is thus a matter of algorithmic optimization. The first proposition, by Dixon et al. [1989], André and Royer [1992], was to minimize the color number; this is the well-known NP-hard *minimum graph coloring* problem [Garey and Johnson 1979]. A first improvement, proposed by Pugh and Weddell [1990] and Ducournau [1991;



A and *B* classes are presumed to have the same implementation as in Fig. 1, page 10, and the diagram follows the same convention. The holes (in grey) in the *C* tables are reserved spaces for *B* in the tables of *D*. In contrast, class *E* can occupy the same positions as *B* because both classes have no common subclass.

Fig. 10. Unidirectional coloring applied to classes, methods and attributes

1997], consists of minimizing the total table size. The tables resulting from coloring are then similar to SST tables, except that they may contain *holes*, i.e. empty entries. The second improvement, by Pugh and Weddell [1990], is *bidirectional* coloring, with positive and negative colors. These new problems were proven to be as difficult as the original problem, therefore heuristics are needed, and some experiments by Pugh and Weddell [1990], Ducournau [1997; 2006] show their efficiency and that these improvements are effective.

The time overhead of multiple inheritance vanishes but holes induce small space overhead. Statistics in Table XII, page App–21, show that the hole rate in the method tables is generally less than 10% and always lower than 40%. For static tables (methods and classes), this overhead is insignificant compared to the cubic table size of standard implementation—compare SMI and COL in Table XII.

Regarding dynamic memory, the overhead is far lower than that of subobjects—see Tables VII, page App–17, and VIII. However, it may be significant; one should actually minimize the total dynamic memory, which requires profiling of class instantiation. Hence, a conservative solution might be to simulate accessors (Section 4.1, page 26) instead of coloring attributes; the offset of each attribute group would be colored in the same way as methods. Dynamic memory overhead disappears to the detriment of a constant time overhead for attribute accesses; it may be reduced by Myers' [1995] double compilation (Section 6.5, page 43).

Overall, coloring gives exactly the same implementation as a SST implementation for a SST hierarchy—this corrects the main drawback of SMI implementation. In case of MI, the overhead vs. SST only concerns static tables and access to some attributes, but this overhead remains far from all other MI implementation overheads (Sections 3 and 4). Interested readers are referred to a review of the approach in [Ducournau 2006]. Note that row displacement [Driesen and Hölzle 1995] would give similar efficiency for method tables and accessor simulation, but

it does not apply directly to attributes. Application to subtype testing has not yet been considered, but it would be a straightforward application of the equivalence of all mechanisms (Section 2.6, page 15).

6.3.2 *Link-Time Coloring.* Like all global layout techniques, coloring only requires the *external schema* of classes and it might be computed at link-time. Although Pugh and Weddell [1990] already noted this possibility, the approach does not seem to have been tested until recently, in the PRM compiler (Section 6.5.2). Furthermore, the choice between attribute coloring and accessor simulation might also be done at link-time, as the code for the former is exactly the same as the efficient version of the code for the latter. Therefore, the linker may choose the best implementation, according to a user-defined overhead threshold.

6.3.3 *Bi- and n-Directional Coloring.* Bidirectionality has been reused in different contexts since this first usage, first in the THETA implementation proposed by Myers [1995] (Section 5.1.3, page 32). In [Eckel and Gil 2000], a positive or negative direction is arbitrarily assigned to classes without superclasses—specializing two classes with opposite direction saves on VBPTRs (Appendix A.3). Gagnon and Hendren [2001] propose a bidirectional layout for optimizing garbage collection (Appendix B.7).

A generalization to *n-directional coloring* has been proposed by [Pugh and Weddell 1993], then independently rediscovered by [Zibin and Gil 2003a]. n can be fixed a priori. For instance, unidirectional ($n = 1$) coloring is usual coloring, when all colors are positive numbers. Bidirectional ($n = 2$) coloring involves positive and negative colors. The hole number decreases as n increases and a *perfect*, i.e. without holes, n-directional coloring can be searched for. However, when $n > 2$, the approach leads to several ($\lceil n/2 \rceil$) independent subobjects, which must be handled with pointer adjustments or with accessor simulation. Hence, it can be considered as an optimization of accessor simulation and Myers' double compilation where the number of efficient accesses is improved.

Overall, bidirectional coloring is likely the best tradeoff, especially at link-time. However, when an object has an actual bidirectional layout, it would certainly facilitate memory management to add at the object head an extra pointer to the method table (Appendix B.7).

6.4 Global Optimizations

Separate compilation allows very few optimizations, with limited applicability (Section 2.5, page 15). Global compilation makes these optimizations more powerful and allows many others.

6.4.1 *Automatic Devirtualization.* Assuming that the C++ `virtual` keyword (Appendix A.1) is only warranted for efficiency reasons, devirtualization computes the optimal subset of `virtual` keywords that are required for avoiding repeated inheritance and minimizing the subobject number [Gil and Sweeney 1999; Eckel and Gil 2000]. More precisely, it determines how to share subobjects. In the diamond example, Figure 3, page 17, it is possible to merge the subobjects D and B, together with the subobjects A and C, thus saving 3 method tables in Table II, page 18. Thus devirtualization involves partitioning $\prec_{\mathrm{d}}$ into two relationships, $\prec_{\mathrm{v}}$

and $\prec_{\mathrm{nv}}$, in such a way that the latter induces diamond-free inheritance (DFI). Once again, this is a matter of global algorithmic optimization.

6.4.2 *Type Analysis.* The main objective of type analysis is to determine the *concrete type* of each program expression, i.e. the set of dynamic types that the expression will take for all possible executions. The main benefit of knowing these concrete types is that the generated code can be adapted to each call site, according to whether the site is *monomorphic* (a single type), *oligomorphic* (a few types) or *megamorphic* (many types). They can be implemented, respectively, by static calls, depth-bounded BTD and coloring. This combination makes BTD time-constant and allows the compiler to restrict method tables to methods that present some megamorphic call sites. A secondary goal of type analysis is to type check programs, in order to save some dynamic tests [Wang and Smith 2001].

Type analysis can be based on the construction of a call graph, but, with object-oriented languages, the two problems are interdependent. A call graph is needed to get concrete types, but a call graph requires concrete types. Without loss of generality, the type analysis problem is exponential (even undecidable as it poses the problem of program termination), but simplifying assumptions make it polynomial [Gil and Itai 1998]. As type analyses always compute an upper bound of concrete types, their accuracy and cost may markedly vary. Grove and Chambers [2001] present a survey of this topic. *Rapid Type Analysis* (RTA) [Bacon and Sweeney 1996] is a classic tradeoff between the simple CHA (Section 6.1.1, page 37) and more accurate algorithms like *Control Flow Analysis* (CFA) [Shivers 1991].

6.4.3 *Dead Code.* An interesting by-product of type analysis is the ability to distinguish between living and dead classes, methods and attributes. Indeed, the call graph associated with a type analysis highlights classes which are never instantiated, methods which are never called and attributes which are never read, in that they are unreachable from the `main` procedure. Type analysis is thus a good way to reduce the code size of applications. However, not all applications will benefit from it. This is not the case, for instance, for applications where class instantiation results from some external interaction especially for languages equipped with a *meta-object protocol* like Java. In this context, all classes are potentially alive.

6.4.4 *Method Copy.* When the source code of superclass methods is known at compile-time, it is possible to copy the code of inherited and not overridden methods into each subclass. The main advantage of this technique, termed *customization* by [Chambers and Ungar 1989], is that `self` becomes monomorphic in each copy; all method calls to `self` can then be compiled into a static call. When the attributes are encapsulated, as in Smalltalk, attribute invariance does not matter and the copy makes attribute offsets static. Otherwise, non-`self` attribute access must be encapsulated by accessors generated or simulated by the compiler (Section 4.1, page 26). Calls to `self` and `super` can also be inlined. Finally, method copy gives more accurate type information, in case of type overriding—for instance, *virtual types* [Torgersen 1998] and Eiffel's *anchored types* can be replaced by constant types. Only accesses to `self` are optimized but the type analysis will propagate this to all call sites. An implementation technique is, however, required for the general case. Moreover, time efficiency is improved to the detriment of static space. Indeed,

method code is duplicated with a factor which is roughly linear in the class number and close to 10 in our benchmarks (Table X, page App–19). Thus this technique cannot be envisaged without being associated with *dead code* elimination. Finally, type analysis opens the way to *code specialization*, a generalization of *customization* at the method [Dean et al. 1995] or at the class level [Tip and Sweeney 2000].

6.4.5 *Inlining.* Inlining is a final optimization of monomorphic calls and, in global compilation, it can also be applied to tree-based techniques. Attribute inlining must also be considered, which consists of replacing an object address by the object itself in the layout of another object, as with the EIFFEL `expanded` keyword. It will save a `load` instruction, but it is only sound under two conditions: type analysis must prove that the attribute is monomorphic, and alias analysis must prove that the same object will not be inlined in two different objects.

6.4.6 *Profiling.* Profiling can be used at various levels. At the call site level, *type feedback* [Agesen and Hölzle 1995] is an improvement to *type prediction*, where expected types are ordered according to their measured frequency. The technique was also used in CEYX [Hullot 1985]. At the object layout level, profiling can count the instances of each class in order to optimize the total number of holes in attribute coloring. *Object splitting* [Chilimbi et al. 1999] has been proposed to improve heap locality and reduce cache and page misses. It consists of splitting the object layout in two subobjects: the first one contains frequently used attributes and points to the second one which contains rarely used attributes.

6.4.7 SMART EIFFEL. The GNU EIFFEL compiler is a typical use of these global techniques in the framework of a statically typed language. It is based on a double a priori, namely global compilation without method tables. In the object layout, the pointer to the method table is replaced by the class identifier. The compiler uses the following techniques: (i) method copy (customization); (ii) type analysis with RTA; (iii) dead code and classes are not compiled; (iv) method calls still polymorphic after step (ii) are implemented with unbounded BTD; the same technique is used for polymorphic accesses to attributes, when the offset varies according to concrete types, as well as for downcasts; (v) finally, many inlinings are done. The recompilation speed is ensured by producing C code, with some optimizations for avoiding useless recompilations of C files. Empirical results show a clear improvement on existing EIFFEL compilers [Zendra et al. 1997; Collin et al. 1997].

6.5 Link-Time Optimizations

Many global optimizations could be applied at link-time after separate compilation. Several approaches can be considered: (i) computation of object representation with mere symbol substitution, as for coloring (Section 6.3, page 39); (ii) multiple separate compilation, with link-time selection; (iii) link-time generation of small pieces of code. All global optimizations are, however, not adapted to such an usage. For instance, devirtualization (Section 6.4.1, page 41) involves both simplifying object representation, that can be done at link-time, and reducing pointer adjustments that must be inlined at compile-time in the generated code.

6.5.1 *Double Compilation.* In this approach, separate compilation generates several versions of the code for each compilation unit. The appropriate version

is chosen at link-time, on the basis of some class hierarchy analysis. Double compilation was proposed by Myers [1995] for optimizing attribute access with accessor simulation (Section 4.1, page 26). It could also apply to method invocation for mixins (Section 5.4, page 34). This implies some global analysis to determine which classes are, in some sense, primary or secondary. Double compilation can be based on other criteria as long as they are invariant by specialization. This is only interesting for optimizing accesses to `self` and receivers typed by the current class.

6.5.2 *Link-Time Generation of Dispatch Code.* In all implementation techniques, the dispatch code is a small piece of code that is generated and inlined at compile-time. It might be generated at link-time instead. In this approach, a method call is compiled into a simple static function call. This function is generated at link-time. It consists of the required code sequence specific to both a given implementation technique and the considered call site, and ends by jumping to the callee address. Therefore, these pieces of code are exactly like *thunks*. Instead of calling a method, the caller calls a thunk which jumps to the method and the resulting overhead is only one jump. The difference with SMI thunks is that static and dynamic calls are inverted here. To gain from this in a global linking setting, some type analysis is required and *Class Hierarchy Analysis* (CHA) is well adapted to this use at link-time, since it relies on the *external schemata* of all classes (Section 6.1.1, page 37). The present approach provides the same kind of benefits as thunks, namely the fact that, in best cases, the thunk is the method itself.

Separate Type Analysis. More accurate link-time type analyses need more information. Type analysis requires source code, but it may be split up into two phases: intra- and inter-class analyses. This is an object-oriented formulation of classic intra- and inter-procedural analyses. Privat and Ducournau [2005] propose to join these two phases by an *internal schema*, produced by intra-class analysis during separate compilation and which stands for an abstract of the class code, specifying the flow of types in the methods. Internal schemata are closely related to the *template* notion proposed by [Agesen 1996]; roughly speaking, the internal schema of a class consists of the templates of its method definitions. At link-time, inter-class analysis uses these internal schemata to construct the call graph and determine concrete types for all expressions in internal schemata. Such an approach would allow the compiler to detect dead code and to remove it from the executable. A similar approach were proposed in a functional language framework [Boucher 2000].

The Prm *Compiler-Linker.* The Prm compiler has been designed as a testbed for various implementation techniques in different compilation schemes, from pure separate to pure global compilation, with some middle points like global linking [Privat and Ducournau 2005]. The linker implements a combination of coloring and bounded BTD, with various type analyses like CHA, RTA and 0-CFA (Section 6.4.2, page 42). Prm code units are compiled into C files that are compiled and linked together, firstly by a dedicated linker that generates all the required thunks, then by the common linker. Early results of these experiments with *all other things being equal* are presented in [Ducournau et al. 2009].

6.6 Load-Time Optimizations

Applying global optimizations at load-time is a great challenge because dynamic loading naturally prefers incremental techniques. Reconciling both approaches involves thus some recompilations. In the following, one assumes that, when a class is loaded: (i) all its superclasses have been previously loaded, (ii) the external schema of all imported classes has already been loaded, otherwise recursive load is possible. Lazy optimizations can be considered, that are, however, beyond the scope of this survey. Overall, loading one class generally means loading a set of related classes.

JIT compilers. Modern virtual machines (VM) are usually equipped with a so-called *just-in-time* (JIT) compiler, that compiles the loaded code or recompiles already loaded code, according to a variety of policies that will not be detailed here. JIT compilers are *adaptive* [Arnold et al. 2005], as they generate the target code according to the current state of the VM. Our simplified view of adaptive compilers is that they compile at load-time under a provisional CWA. Hence, any global optimization can apply, with the proviso that it does not yield too heavy recomputations and recompilations. It can be based, for instance, on a simple class hierarchy analysis (CHA, Section 6.1.1, page 37), with the hierarchy being restricted to already loaded classes. When loading a class, each call site in the loaded code can be recognized by CHA as either already polymorphic, or currently monomorphic. In the case of interface-typed receiver (Section 5, page 30), the compiler can distinguish a third level, when the interface is directly implemented by a single class—hence, the call can be compiled as a call on a class-typed receiver. Each call site can thus be compiled under three forms: (i) a static call, (ii) a general class or (iii) interface method invocation. When subsequent class loadings will invalidate the assumption, some methods will need to be recompiled, and some call sites will change.

Thunks for Optimized Calls. The previously described link-time optimizations can be adapted to JIT compilers in order to avoid full method recompilations. When a method is compiled at load-time, unoptimized call sites are compiled as usual and optimized call sites are compiled into a static call to a dispatch function (i.e. a *thunk*) which can be shared with all similar call sites. For a monomorphic call, the thunk jumps to the method address. For an interface that is directly implemented by a single class, the thunk implements a class invocation method. At the time of a further class loading, previously compiled call sites may change and the thunks must be updated. The VM must maintain a class representation that associates each thunk with the concerned class and method. When the loaded class overrides some method, the compiler must only search for concerned thunks in the superclasses and recompile them. This should be markedly less costly than recompiling full methods. The survey by Arnold et al. [2005] does not mention any similar technique but one can find, on the web, mentions of "trampoline method calls" in Berndt Mathiske's Maxine VM. The overhead at load-time does not seem to be important, since all statistics show that most call sites are monomorphic so the approach would require very few recompilations, and each one would be very small. However, the run-time gain remains speculative because of the hypothetical cost of these thunks and experiments are required.

Incremental Coloring. Global techniques such as coloring are intrinsically non-incremental. When the loaded class has several superclasses, it may introduce conflicts between the superclass colors, which are difficult to solve and propagate. Enlarging method tables or allocating new ones is not possible; therefore the implementation must rely on extra indirections, with the color table being pointed to by the method table. In that way, it is possible to reallocate color tables when they must be enlarged. Thus, incremental coloring is possible but quite intricate. Palacz and Vitek [2003] propose to use it for subtyping tests when the target is an interface. The equivalence of method invocation and subtype testing (Section 2.6) makes it possible to broaden the approach to method invocation [Ducournau 2008], in the same way as for perfect hashing (Section 4.2, page 27). This implements method invocations and subtype tests with only some extra indirections. This is an acceptable run-time overhead, but the main issue is at load-time, since optimal coloring is NP-hard and the heuristics are cubic in the class number.

6.7 Conclusion on Global Approaches

Several compilation schemes are to be considered under the CWA. With global compilation, a mixing of coloring and bounded BTD, coupled with type analysis and dead code elimination, certainly provides an optimal implementation basis. Global compilation obviously affords the best run-time efficiency but it also has significant drawbacks from the proprammer's standpoint: (i) it does not allow the programmer to check the safety of a single piece of code, and only provides *system-level safety*; (ii) with dead-code elimination, only the living code is actually compiled and the dead code may not even be checked for the sake of compile-time efficiency; this can puzzle programmers, who may think that their last change has been checked. Of course, there are solutions to these drawbacks, but they might reduce the compile-time efficiency.

Anyway, leaving the modularity provided by separate compilation may be considered as too high a price for program optimization. Link-time coloring provides an efficient implementation—markedly better than what can be obtained under the OWA without any recompilation—which has the advantages of separate compilation and keeps the overall architecture simple. Other link-time optimizations are feasible but our early results do not allow us to conclude whether the runtime gain offsets the architecture complexity. The application of these link-time optimizations to adaptive compilers requires further tests. Finally, JIT compilers are not, in theory, incompatible with standard MI implementation—however, both are quite intricate and their combination would be overly complicated. Therefore, an hypothetical runtime system with full multiple inheritance and dynamic loading would certainly benefit from alternative implementations.

7. CONCLUSION AND PROSPECTS

7.1 Conclusions

Different conclusions can be drawn from this survey, according to whether one stresses language expressivity, namely multiple vs. single inheritance, or runtime system flexibility, namely dynamic loading vs. global compilation or linking.

On the one hand, separate compilation of single-subtyping (SST) is simple and

as efficient as possible. Indirect method calls are true overhead which could only be reduced with global optimizations or by increasing the processor capabilities for indirect branching prediction [Driesen 2001]. But SST expressiveness is far from what programmers could expect and, as far as we know, there is no commonly used SST language. On the other hand, separate compilation of full multiple inheritance (MI) presents significant overhead with respect to SST, and the main drawback of the standard implementation is that it is as costly when MI is not used. Another drawback, explicit both in its cubic worst-case and through benchmark measurement (see Appendix C), is its poor scalability. Therefore, it is not surprising that recent efforts have been focused on multiple-subtyping (MST) languages, like JAVA or C#. This is a sound middle point between the two extremes, especially when compared to other tradeoffs such as non-virtual inheritance (NVI) or mixins. Although they are among the most used languages, JAVA and C# represent several implementation issues: interfaces, boxing and generics (the latter only for JAVA). An efficient implementation of interfaces is almost as difficult as that of full multiple inheritance and programming usage can imply intensive use of interfaces. Hence, the efficiency must be as high for interfaces as for classes and its scalability must be assessed in the worst case. These conclusions are drawn irrespective of the optimizations that might be provided by adaptive compilers. Indeed, an efficient basic implementation is required for cases where no specific optimization apply.

A general solution to the efficiency issue presented by object-oriented programming is global compilation, under full CWA, which is *necessarily* more efficient than any other compilation scheme, since it includes all of them—indeed, any technique can be used in a global compilation setting. However, global compilation has a crippling drawback, namely its lack of modularity. There is thus a tradeoff between efficiency and modularity, and it would seem that there are two ways of solving this tradeoff, that both imply some mixing of OWA and CWA. For JAVA and .NET platforms, the language specifications are based on full OWA and dynamic loading, but the runtime system relies on provisional CWA that provides efficient implementation at the expense of some recompilations that are expected to be both unfrequent and cheap. In the PRM experimental compiler-linker, the language specifications do not require the OWA, but the compiler is fully compatible with it, while the linker implies the CWA. Many variants are possible that mix separate and global compilations and can be thought of as modular ways of using global compilation. The resulting efficiency is midway between fully separate and global compilations. It is certainly worthwhile to consider this approach when dynamic loading is not required, but a precise comparison with global compilation would require further experiments. The point would be to determine whether the gain in global compilation over the mixed approach is significant enough to offset its drawbacks. Finally, besides dynamic loading that may be a functional requirement, it would be interesting to compare the respective efficiencies of these two compromises between open and closed world assumptions.

7.2 Prospects

The prospects of this work are three-fold and concern (i) more in-depth experiments, (ii) current production runtime systems, and (iii) further research.

More Experiments. In this survey, we have proposed an evaluation of the presented techniques. This evaluation relies on an abstract computation model that gives a rough idea of the respective time efficiencies. Space efficiency has been assessed through accurate simulation of the memory occupation of programs consisting of large scale class hierarchies (Appendix C). These simulations are exact but they do not represent actual programs. Hence, they likely account more for the scalability of the different techniques than for the efficiency of current programs. Therefore, large-scale experiments on real programs are mandatory. We are currently running a set of experiments on the Prm testbed (Section 6.5.2, page 44) to compare a variety of implementation techniques and compilation schemes with *all other things being equal*. The results of these experiments are too early and not complete enough to be presented here. Interested readers are referred to [Ducournau et al. 2009] and forthcoming papers for a systematic assessment of the runtime efficiency of the various techniques that we have described in this survey. All techniques that apply to full MI can be tested in this testbed. Therefore, only the techniques that are specific to MST and do not apply to full MI, e.g. the subobject alternative for interface implementation (Section 5.1, page 30), cannot be tested. All other techniques that apply to MST can be assessed by extrapolating their efficiency in a full MI inheritance setting, with the considered technique (e.g. perfect hashing) applying to method invocation and subtype testing, whereas attribute access uses attribute coloring. This makes the experimentation more demanding than in actual MST settings, since the tested implementation is used here for all method invocations and subtype tests, instead of being restricted to interfaces. The testbed is also inadapted to dynamic loading. Experiments in actual virtual machines are thus required for precise assessment of load-time dynamic effects.

Production Runtime Systems. This article stressed several points where an improvement is possible in the present state of affairs. There is some evidence that standard MI implementation might be improved with *empty-subobject optimization*. According to our benchmarks, this would markedly reduce the overhead of C++ virtual inheritance. The implementation of Java-like interfaces also represents an issue in current runtime systems whose worst-case efficiency does not seem to be ensured. It would seem that *perfect hashing* responds pretty well to this issue, according to both our abstract analysis and the aforementioned experiments with the Prm testbed. This should be confirmed by experiments on production VM. Perfect hashing would also represent an efficient implementation of downcasts in all languages with full MI.

Further Research. There remains a major issue at the end of this survey. An efficient incremental implementation of multiple inheritance—by efficient, we mean better than standard subobject-based MI—is certainly the point where a significant improvement will be the most difficult to achieve. Incremental coloring and perfect hashing are likely not more efficient than subobject-based implementations from the time standpoint, because of attribute access. A clean integration of primitives types, as in C#, Eiffel and Java but unlike C++, would even increase the challenge. A dual open issue would be to design an efficient virtual machine based on full MI specifications, with an adaptive compiler dedicated to multiple inheritance. For

instance, Myers' double compilation could be used for optimizing attribute access. The goal might be to reach the same efficiency as that of JAVA and .NET on a class hierarchy where classes that introduce attributes form a tree.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/csur/2010-42-N/p1-Ducournau`.

ACKNOWLEDGMENTS

This work was partially supported by grants from Région Languedoc-Roussillon (034750). The author thanks Yoav Zibin for reading of a preliminary version and for valuable comments.

REFERENCES

AGESEN, O. 1996. Concrete type inference: Delivering object-oriented applications. Ph.D. thesis, Stanford University.

AGESEN, O. AND HÖLZLE, U. 1995. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *Proc. OOPSLA'95*. SIGPLAN Not. 30(10). ACM, 91–107.

ALPERN, B., ATTANASIO, C., BARTON, J., COCCHI, A., HUMMEL, S., LIEBER, D., NGO, T., MERGEN, M., SHEPHERD, J., AND SMITH, S. 1999. Implementing Jalapeño in Java. In *Proc. OOPSLA'99*. SIGPLAN Not. 34(10). ACM, 314–324.

ALPERN, B., COCCHI, A., FINK, S., AND GROVE, D. 2001a. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proc. OOPSLA'01*. SIGPLAN Not. 36(10). ACM, 108–124.

ALPERN, B., COCCHI, A., AND GROVE, D. 2001b. Dynamic type checking in Jalapeño. In *Proc. USENIX JVM'01*.

ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2003. Jam—designing a java extension with mixins. *ACM Trans. Program. Lang. Syst. 25,* 5, 641–712.

ANDRÉ, P. AND ROYER, J.-C. 1992. Optimizing method search with lookup caches and incremental coloring. In *Proc. OOPSLA'92*. SIGPLAN Not. 27(10). ACM, 110–126.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. 2005. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE 93,* 2 (Feb.), 449–466.

BACON, D. AND SWEENEY, P. 1996. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*. SIGPLAN Not. 31(10). ACM, 324–341.

BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., MOSS, B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA'06*, P. L. Tarr and W. R. Cook, Eds. SIGPLAN Not. 41(10). ACM, 169–190.

BOEHM, H.-J. 1993. Space-efficient conservative garbage collection. In *Proc. PLDI'93*. SIGPLAN Not. 28(6). ACM, 197–206.

BOUCHER, D. 2000. GOld: a link-time optimizer for Scheme. In *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, M. Felleisen, Ed. 1–12.

BOYLAND, J. AND CASTAGNA, G. 1996. Type-safe compilation of covariant specialization: a practical case. In *Proc. ECOOP'96*, P. Cointe, Ed. LNCS 1098. Springer, 3–25.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *Proc. OOPSLA/ECOOP'90*. SIGPLAN Not. 25(10). ACM, 303–311.

BURTON, K. 2002. *.NET Common Language Runtime – Unleashed*. Sams.

Calder, B. and Grunwald, D. 1994. Reducing indirect function call overhead in C++ programs. In *Proc. POPL'94*. ACM, 397–408.

Cargill, T. A. 1991. Controversy: The case against multiple inheritance in C++. *Computing Systems 4,* 1, 69–82.

Castagna, G. 1997. *Object-oriented programming: a unified foundation*. Birkhaüser.

Chambers, C. and Ungar, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *Proc. OOPSLA'89*. SIGPLAN Not. 24(10). ACM, 146–160.

Chen, W., Turau, V., and Klas, W. 1994. Efficient dynamic look-up strategy for multi-methods. In *Proc. ECOOP'94*, M. Tokoro and R. Pareschi, Eds. LNCS 821. Springer, 408–431.

Chilimbi, T. M., Davidson, B., and Larus, J. R. 1999. Cache-conscious structure definition. In *Proc. PLDI'99*. SIGPLAN Not. 34(5). ACM, 1–12.

Click, C. and Rose, J. 2002. Fast subtype checking in the Hotspot JVM. In *Proc. ACM-ISCOPE Conf. on Java Grande (JGI'02)*. 96–107.

Cohen, N. H. 1991. Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst. 13,* 4, 626–629.

Collin, S., Colnet, D., and Zendra, O. 1997. Type inference for late binding. the SmallEiffel compiler. In *Proc. Joint Modular Languages Conference*. LNCS 1204. Springer, 67–81.

Cook, W. R. 1989. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89*, S. Cook, Ed. Cambridge University Press, 57–70.

Czech, Z. J., Havas, G., and Majewski, B. S. 1997. Perfect hashing. *Theor. Comput. Sci. 182,* 1-2, 1–143.

Dean, J., Chambers, C., and Grove, D. 1995. Selective specialization for object-oriented languages. In *Proc. PLDI'95*. SIGPLAN Not. 30(6). ACM, 93–102.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. ECOOP'95*, W. Olthoff, Ed. LNCS 952. Springer, 77–101.

Dijkstra, E. W. 1960. Recursive programming. *Numer. Math. 2*, 312–318.

Dixon, R., McKee, T., Schweitzer, P., and Vaughan, M. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*. SIGPLAN Not. 24(10). ACM, 211–214.

Driesen, K. 2001. *Efficient Polymorphic Calls*. Kluwer Academic Publisher.

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. In *Proc. OOPSLA'95*. SIGPLAN Not. 30(10). ACM, 141–155.

Driesen, K. and Hölzle, U. 1996. The direct cost of virtual function calls in C++. In *Proc. OOPSLA'96*. SIGPLAN Not. 31(10). ACM, 306–323.

Driesen, K., Hölzle, U., and Vitek, J. 1995. Message dispatch on pipelined processors. In *Proc. ECOOP'95*, W. Olthoff, Ed. LNCS 952. Springer, 253–282.

Driesen, K. and Zendra, O. 2002. Stress-testing control structures for dynamic dispatch in Java. In *Proc. Java Virtual Machine Research and Technology Symp., JVM'02*. Usenix, 105–118.

Ducournau, R. 1991. *Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual*. Sema Group, Montrouge, France.

Ducournau, R. 1997. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet 3,* 3, 241–276.

Ducournau, R. 2006. Coloring, a versatile technique for implementing object-oriented languages. Tech. Rep. LIRMM-06001, Université Montpellier 2.

Ducournau, R. 2008. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst. 30,* 6, 1–56.

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M.-L. 1994. Proposal for a monotonic multiple inheritance linearization. In *Proc. OOPSLA'94*. SIGPLAN Not. 29(10). ACM, 164–175.

Ducournau, R. and Morandat, F. 2009. More results on perfect hashing for implementing object-oriented languages. Tech. Rep. LIRMM-09001, Université Montpellier 2.

Ducournau, R., Morandat, F., and Privat, J. 2009. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In *Proc. OOPSLA'09*, G. T. Leavens, Ed. SIGPLAN Not. 44(10). ACM, 41–60.

Ducournau, R. and Privat, J. 2008. Metamodeling semantics of multiple inheritance. Tech. Rep. LIRMM-08017, Université Montpellier 2.

Dujardin, E., Amiel, E., and Simon, E. 1998. Fast algorithms for compressed multimethod dispatch table generation. *ACM Trans. Program. Lang. Syst. 20,* 1, 116–165.

Eckel, N. and Gil, J. 2000. Empirical study of object-layout and optimization techniques. In *Proc. ECOOP'2000*, E. Bertino, Ed. LNCS 1850. Springer, 394–421.

Ellis, M. and Stroustrup, B. 1990. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US.

Ernst, E. 2002. Safe dynamic multiple inheritance. *Nord. J. Comput 9,* 1, 191–208.

Fähndrich, M. and Leino, K. R. M. 2003. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA'03*, R. Crocker and G. L. S. Jr., Eds. SIGPLAN Not. 38(11). ACM, 302–312.

Gagnon, E. M. and Hendren, L. J. 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *Proc. USENIX JVM'01.* 27–40.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco (CA), USA.

Gil, J. and Itai, A. 1998. The complexity of type analysis of object oriented programs. In *Proc. ECOOP'98.* LNCS 1445. Springer, 601–634.

Gil, J. and Sweeney, P. 1999. Space and time-efficient memory layout for multiple inheritance. In *Proc. OOPSLA'99.* SIGPLAN Not. 34(10). ACM, 256–275.

Gil, J. and Zibin, Y. 2005. Efficient subtyping tests with PQ-encoding. *ACM Trans. Program. Lang. Syst. 27,* 5, 819–856.

Gil, J. and Zibin, Y. 2007. Efficient dynamic dispatching with type slicing. *ACM Trans. Program. Lang. Syst. 30,* 1, 5:1–53.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading (MA), USA.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The JAVA Language Specification*, Third ed. Addison-Wesley.

Grove, D. and Chambers, C. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23,* 6, 685–746.

Holst, W., Szafron, D., Leontiev, Y., and Pang, C. 1998. Multi-method dispatch using single-receiver projections. Technical Report TR-98-03, University of Alberta, Edmonton, ca.

Hölzle, U., Chambers, C., and Ungar, D. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP'91*, P. America, Ed. LNCS 512. Springer, 21–38.

Huchard, M. and Leblanc, H. 2000. Computing interfaces in Java. In *Proc. of IEEE Int. Conf. on Automated Software Engineering (ASE'2000).* 317–320.

Hullot, J.-M. 1985. Ceyx version 15. Technical Report 44-46, INRIA.

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. 1997. Back to the future: The story of Squeak - a usable Smalltalk written in itself. In *Proc. OOPSLA'97.* SIGPLAN Not. 32(10). ACM, 318–326.

Jones, R. and Lins, R. 1996. *Garbage Collection.* Wiley.

Jul, E. 2008. Precomputing method lookup. In *Workshop ICOOOLPS at ECOOP'08.*

Kennedy, A. and Syme, D. 2001. Design and implementation of generics for the .NET Common Language Runtime. In *Proc. PLDI'01.* SIGPLAN Not. 36(5). ACM, 1–12.

Kiczales, G. and Rodriguez, L. 1990. Efficient method dispatch in PCL. In *Proc. ACM Conf. on Lisp and Functional Programming.* 99–105.

Knuth, D. E. 1973. *The art of computer programming, Sorting and Searching.* Vol. 3. Addison-Wesley.

Koenig, A. 1998. Standard – the C++ language. Report ISO/IEC 14882:1998, Information Technology Council (NCTIS). `http://www.nctis.org/cplusplus.htm`.

Krall, A. and Grafl, R. 1997. CACAO - a 64 bits JavaVM just-in-time compiler. *Concurrency: Practice ans Experience 9,* 11, 1017–1030.

Krogdahl, S. 1985. Multiple inheritance in Simula-like languages. *BIT 25,* 2, 318–326.

Lippman, S. B. 1996. *Inside the C++ Object Model*. New York.

Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P., and Myers, A. C. 1995. THETA reference manual. Technical report, MIT.

Meyer, B. 1992. *Eiffel: The Language*. Prentice-Hall.

Meyer, B. 1997. *Object-Oriented Software Construction*, second ed. Prentice-Hall.

Meyer, J. and Downing, T. 1997. *JAVA Virtual Machine*. O'Reilly.

Microsoft. 2001. C# Language specifications, v0.28. Technical report, Microsoft Corporation.

Mugridge, W. B., Hamer, J., and Hosking, J. G. 1991. Multi-methods in a statically-typed programming language. In *Proc. ECOOP'91*, P. America, Ed. LNCS 512. Springer, 307–324.

Muthukrishnan, S. and Muller, M. 1996. Time and space efficient method lookup for object-oriented languages. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*. ACM/SIAM, 42–51.

Myers, A. 1995. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*. SIGPLAN Not. 30(10). ACM, 124–139.

Odersky, M., Spoon, L., and Venners, B. 2008. *Programming in Scala, A comprehensive step-by-step guide*. Artima.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Proc. POPL'97*. ACM, 146–159.

Palacz, K. and Vitek, J. 2003. Java subtype tests in real-time. In *Proc. ECOOP'2003*, L. Cardelli, Ed. LNCS 2743. Springer, 378–404.

Privat, J. and Ducournau, R. 2005. Link-time static analysis for efficient separate compilation of object-oriented languages. In *ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05)*. 20–27.

Pugh, W. and Weddell, G. 1990. Two-directional record layout for multiple inheritance. In *Proc. PLDI'90*. SIGPLAN Not. 25(6). ACM, 85–91.

Pugh, W. and Weddell, G. E. 1993. On object layout for multiple inheritance. Tech. Rep. CS-93-22, University of Waterloo.

Queinnec, C. 1998. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters 64,* 6, 315–321.

Raynaud, O. and Thierry, E. 2001. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proc. ECOOP'2001*, J. L. Knudsen, Ed. LNCS 2072. Springer, 165–180.

Sakkinen, M. 1989. Disciplined inheritance. In *Proc. ECOOP'89*, S. Cook, Ed. Cambridge University Press, 39–58.

Sakkinen, M. 1992. A critique of the inheritance principles of C++. *Computing Systems 5,* 1, 69–110.

Schubert, L., Papalaskaris, M., and Taugher, J. 1983. Determining type, part, color and time relationship. *Computer 16*, 53–60.

Shang, D. L. 1996. Are cows animals? `http://www.visviva.com/transframe/papers/covar.htm`.

Shivers, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon University.

Smaragdakis, Y. and Batory, D. 2002. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Soft. Eng. Meth. 11,* 2, 215–255.

Sprugnoli, R. 1977. Perfect hashing functions: a single probe retrieving method for static sets. *Comm. ACM 20,* 11, 841–850.

Steele, G. L. 1990. *Common Lisp, the Language*, Second ed. Digital Press.

Stefik, M. and Bobrow, D. 1986. Object-oriented programming: Themes and variations. *AI Magazine 6,* 4, 40–62.

Stroustrup, B. 1998. *The C++ programming Language, 3^e ed.* Addison-Wesley.

Sweeney, P. F. and Burke, M. G. 2003. Quantifying and evaluating the space overhead for alternative C++ memory layouts. *Softw., Pract. Exper. 33,* 7, 595–636.

Taft, S. T., Duff, R. A., Brukardt, R. L., Ploedereder, E., and Leroy, P., Eds. 2006. *Ada 2005 Reference Manual: Language and Standard Libraries.* LNCS 4348. Springer.

Tarjan, R. E. and Yao, A. C. C. 1979. Storing a sparse table. *Comm. ACM 22,* 11, 606–611.

Tip, F. and Sweeney, P. F. 2000. Class hierarchy specialization. *Acta Informatica 36,* 12, 927–982.

Torgersen, M. 1998. Virtual types are statically safe. In *Elec. Proc. of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL 5).*

Ungar, D. and Smith, R. 1987. SELF: The power of simplicity. In *Proc. OOPSLA'87.* SIGPLAN Not. 22(12). ACM, 227–242.

VanHilst, M. and Notkin, D. 1996. Using role components to implement collaboration-based designs. In *Proc. OOPSLA'96.* SIGPLAN Not. 31(10). ACM, 359–369.

Vitek, J. and Horspool, R. N. 1994. Taming message passing: efficient method look-up for dynamically typed languages. In *Proc. ECOOP'94*, M. Tokoro and R. Pareschi, Eds. LNCS 821. 432–449.

Vitek, J., Horspool, R. N., and Krall, A. 1997. Efficient type inclusion tests. In *Proc. OOPSLA'97.* SIGPLAN Not. 32(10). ACM, 142–157.

Vitter, J. S. and Flajolet, P. 1990. Average-case analysis of algorithms and data structures. In *Algorithms and Complexity*, J. Van Leeuwen, Ed. Handbook of Theoretical Computer Science, vol. 1. Elsevier, Amsterdam, Chapter 9, 431–524.

Waldo, J. 1991. Controversy: The case for multiple inheritance in C++. *Computing Systems 4,* 2, 157–171.

Wang, T. and Smith, S. 2001. Precise constraint-based type inference for Java. In *Proc. ECOOP'2001*, J. L. Knudsen, Ed. LNCS 2072. Springer, 99–117.

Weber, F. 1992. Getting class correctness and system correctness equivalent — how to get covariant right. In *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, R. Ege, M. Singh, and B. Meyer, Eds. 192–213.

Wilson, P. R. 1992. Uniprocessor garbage collection techniques. In *Int. Workshop on Memory Management (IWMM'92).* LNCS 637. Springer, 1–42.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proc. OOPSLA'97.* SIGPLAN Not. 32(10). ACM, 125–141.

Zibin, Y. and Gil, J. 2002. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proc. OOPSLA'02.* SIGPLAN Not. 37(10). ACM, 142–160.

Zibin, Y. and Gil, J. 2003a. Incremental algorithms for dispatching in dynamically typed languages. In *Proc. POPL'03.* SIGPLAN Not. 38(1). ACM, 126–138.

Zibin, Y. and Gil, J. 2003b. Two-dimensional bi-directional object layout. In *Proc. ECOOP'2003*, L. Cardelli, Ed. LNCS 2743. Springer, 329–350.

Received November 2002; revised July 2005, August 2008; accepted March 2009

Table IV. Index of Acronyms

	definition	Section	page
AS	accessor simulation	4.1	26
BTD	binary tree dispatch	6.2.2	38
CFA	control flow analysis	6.4.2	42
CHA	class hierarchy analysis	6.1.1	37
COL	coloring	6.3	39
CWA	closed-world assumption		
DFI	diamond-free inheritance, when the superclasses of any class form a bottom-up tree	A.1	App–1
DVI	devirtualization, i.e. optimal balance of virtual and non-virtual inheritance	6.4.1	41
ESO	empty-subobject optimization	3.3	23
JIT	just-in-time compilers	6.6	45
MI	multiple inheritance, a class can have several direct superclasses (the hierarchy is a dag)		
MST	multiple subtyping, i.e. single class and multiple interface inheritance, as in JAVA	5	30
NVI	non-virtual inheritance	A.1	App–1
OO	object-oriented, i.e. all that concerns the dynamic type of objects		
OWA	open-world assumption		
PH	perfect hashing	4.2	27
RTA	rapid type analysis	6.4.2	42
SI	single inheritance, that is, a class has a single direct superclass (the hierarchy forms a top-down tree or forest)		
SMI	standard or subobject-based multiple inheritance implementation	3	17
SST	single subtyping, that is, single class inheritance when all types are classes (no JAVA-like interfaces)	2	9
VBPTR	virtual base pointer, i.e. pointers to subobjects in the object layout	A.3	App–6
VFT	virtual functions tables, i.e. method tables in C++ jargon		
VM	virtual machine		

This document is the online-only appendix to:

Implementing Statically Typed Object-Oriented Programming Languages

ROLAND DUCOURNAU
LIRMM – CNRS et Université Montpellier II, France

ACM Computing Surveys, Vol. 42, No. N, 2010, Pages 1–53.

References to Sections refer to the main article, and references to Tables refer to these appendices.

A. SUBOBJECT VARIANTS

This appendix examines small variations of the standard subobject-based implementation (SMI) presented in Section 3, page 17, in order to convince readers that this "standard" is a good choice. These variations consist of a MI implementation without overhead when used only in SI (Appendix A.1), or various tradeoffs between time and space efficiencies (Appendices A.2 and A.3). Finally, the relation between object layout and method tables can rely on table flow in parallel of data flow, instead of explicit pointers (Appendix A.4).

A.1 Non-Virtual Inheritance (NVI)

The main aforementioned drawback of standard MI implementation is that its overhead does not depend on effective use of MI. C++ *non-virtual* inheritance is an attempt to answer this problem. It is syntactically expressed by omitting the keyword `virtual` when declaring superclasses. The main concern with non-virtual inheritance is that it is actually not just an implementation but another semantics of multiple inheritance. The resulting implementation is more efficient, but it is soundly reusable only for *diamond-free inheritance* (DFI), i.e. when the hierarchy does not include any diamond and the superclasses of any class thus form a bottom-up tree. This is what Sakkinen [1989; 1992] calls *independent multiple inheritance*. This implementation was proposed by Krogdahl [1985] in this restricted case only, but it is used in C++ in an unrestricted way.

A.1.1 *Pure NVI.* NVI is better described by its implementation [Ellis and Stroustrup 1990; Lippman 1996] than by its semantics. A root is implemented as in SST. When a class has one or more direct superclasses, the method table and instance layout respectively consist of the concatenation of the corresponding tables of all direct superclasses. The attributes and methods introduced by the class are then added at the end of the respective tables (Figure 12-a), like in SST. Thus, the first difference with respect to SMI is that a non-root class has no proper subobject for τ_d. This implicitly defines a new relation $\prec_{\mathrm{e}}$, a subset of $\prec_{\mathrm{d}}$, between the currently

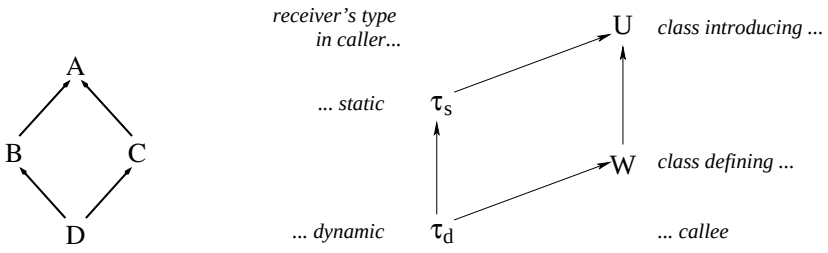


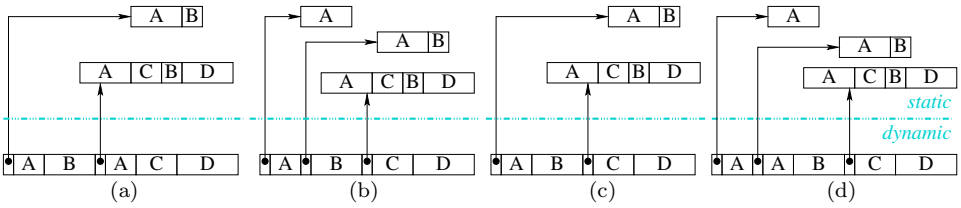
Fig. 11. The multiple inheritance diamond (from Fig. 3, page 17), and the classes involved in late binding (from Fig. 4, page 19)

defined class and the direct superclass whose subobject is shared. Moreover, all $\Delta_{T,U}$ are now invariant w.r.t. τ_d, so upcasts and attribute accesses are truly static as they do not need table access. The code generated for method calls is the same as with SMI but shifts on the receiver are null when the receiver static types in the caller and callee share the same subobject—so thunks make most of them disappear in practice. Moreover, *thunk elimination* is now possible because the shift on the receiver is independent of τ_d, i.e. the overriding method defined in class W (see Figure 11-right) expects a value of `self` typed by the class U introducing the method and two static shifts are made in the caller (from τ_s to U) and in the callee (from U to W). Alternatively, general support for multiple entry points avoids explicit thunks—an entry point is generated for each V such that $W \preceq V \preceq U$. Interested readers are referred to [Lippman 1996, p. 138] for further details.

The layout is a direct generalization of SST. When a class hierarchy is a tree, the layout is exactly the same as with SST, with a single subobject per object and without any non-null shift. In the general case of diamond-free hierarchies, the number of subobjects (or method tables) is equal to the number of *root* superclasses.

A.1.2 *Diamond-Free vs. Repeated Inheritance.* The flaw of non-virtual inheritance occurs when the hierarchy is no longer diamond-free. When the inheritance graph contains undirected cycles (like the diamond $ABCD$ in Figure 11), NVI involves what we call *repeated inheritance*[1], in the sense that some subobject is repeated in the object layout. In Figure 12-a, this is the case for A which is present in both subobjects of B and C—hence, accessing an attribute introduced by A, or a method defined in A, requires an explicit upcast to B or C. In the worst case of repeated inheritance, the number of subobjects becomes exponential in the number of superclasses, as it represents the number of paths from the considered class to its root superclasses. Consider a chain of n diamonds $A_iB_iC_iD_i$ such that $D_i = A_{i+1}$; then A_1 will be repeated 2^n times in A_n layout. However, the point at issue is not only implementation, since programmers have to syntactically distinguish between this exponential number of interpretations. Thus this cannot be considered as a sensible feature, and it is indeed common opinion that "repeated inheritance is an abomination" [Zibin and Gil 2003a, note 2]. Therefore, non-virtual inheritance should be used only on class hierarchies that do not involve repeated inheritance.

[1]Meyer [1997] uses the term to denote undirected cycles in class hierarchies.



(a) pure NVI and repeated inheritance; (b) mixed NVI when $D \prec_{\mathrm{nv}} B, C \prec_{\mathrm{v}} A$; (c) in ideal and (d) actual implementations when $B \prec_{\mathrm{nv}} A$ and $C \prec_{\mathrm{v}} A$. In all cases, $D \prec_{e} C$, i.e. C and D share the same subobject.

Fig. 12. Non-virtual inheritance on the diamond from Fig. 11

A.1.3 *Mixing Virtual and Non-Virtual Inheritance.* This is the usual programming style in C++. Inheritance edges, i.e. $\prec_{\mathrm{d}}$, are partitioned into two sets, $\prec_{\mathrm{v}}$ and $\prec_{\mathrm{nv}}$, according to whether the keyword `virtual` is used or not. The layout of a class C is made of two parts. The non-virtual part NV_C is computed as for pure NVI, but the concatenation on direct superclasses is computed on the $\prec_{\mathrm{nv}}$ relationship, rather than on $\prec_{d}$ (thus $\prec_{\mathrm{e}}$ is a subset of $\prec_{\mathrm{nv}}$):

$$NV_C = \big(\sum_{C \prec_{\mathrm{nv}} D} NV_D \big) + attr_C \tag{8}$$

where $attr_C$ denotes attributes introduced in class C and sum denotes concatenation. When there is no such D, NV_C is like a root in SST implementation. The virtual part V_C is a set of virtually inherited non-virtual parts. On the whole, the object layout consists of *subobject groups*—each group is made of subobjects in a fixed relative position, whereas groups have arbitrary relative positions. However, there are at least two ways of specifying the virtual part since, with the diamond situation in Figure 11, mixing may be ambiguous, e.g. when class A is non-virtual in some superclass, and virtual in another one.

Ideal Implementation. In an ideal specification, NVI would be used only for efficiency. Therefore, when a class is inherited through virtual and non-virtual edges, only one implementation should be used (Figure 12-c). Moreover, $\prec_{\mathrm{nv}}$ should not imply repeated inheritance, so all partitions of $\prec_{\mathrm{d}}$ are not sound. Hence, the virtual part V_C is made of the non-virtual parts of all superclasses, either direct or indirect, which are not reachable through $\prec_{\mathrm{nv}}$ paths, either from C (i.e. which are not already included in NV_C) or from superclasses of V_C (i.e. which are not already included in NV_C or in some element of V_C):

$$V_C = min_{\prec^*_{\mathrm{nv}}} \{ NV_D \mid C \prec D \ \texttt{and}\ \mathrm{C} \not\prec^*_{\mathrm{nv}} D \} \tag{9}$$

where $\prec^*_{\mathrm{nv}}$ is the transitive closure of $\prec_{\mathrm{nv}}$.

Actual C++ Specification and Implementation. C++ actually differs from this ideal implementation. Although `virtual` is an annotation of inheritance edges, the C++ object model makes it an annotation of implemented classes, i.e. subobjects. Hence, when a class is used as both a virtual and nonvirtual superclass, it will have two distinct implementations. Thus, as far as one can extrapolate example-based specifications [Koenig 1998, p. 165], the virtual part V_C is the union of all

virtual parts of superclasses, plus the set of non-virtual parts of direct superclasses inherited through $\prec_{\mathrm{v}}$, including those that are already in the virtual part:

$$V_C = \{NV_D \mid C \prec_{\mathrm{v}} D\} \cup \bigcup_{C \prec_{\mathrm{d}} D} V_D \tag{10}$$

Therefore, a superclass inherited twice, through virtual and non-virtual edges, is repeated (Fig. 12-d) and both links must be virtual in order to avoid repeated inheritance (Fig. 12-b).

A.1.4 *Casting.* With diamond-free hierarchies and pure NVI, the relative positions of two subobjects whose types are related by subtyping do not depend on the dynamic type. Thus upcasts are unambiguous and shifts static, without table access, and often null. As $\Delta_{\tau_s,T}$ is independent of τ_d, downcasts can also be made in a way that is very close to Cohen's display (Section 2.3.2, page 12). Each subobject implements a table of the class ID's corresponding to the subobject and the position of id_T w.r.t. τ_s subobject is constant. Only side casts need the standard MI technique.

On the other hand, repeated inheritance makes casting ambiguous. For upcasts, the ambiguity is on the target and intermediate upcasts may be needed. As for downcasts, the ambiguity is on the source and checks must be repeated as many times as the source is repeated. Side and dynamic casts are not always possible.

Mixing virtual and non-virtual inheritance makes the problem even more complicated. [Ellis and Stroustrup 1990, section 10.6c] prohibits some cast cases. However, according to the specifications of `dynamic_cast` [Koenig 1998], the effective implementation seems to use both $\Delta^{\tau_s}_{\downarrow}$ (in case of repeated inheritance) and $\Delta^{\Uparrow}$ (for sidecast). If the target is unambiguous in one of the two tables, the cast succeeds, i.e. no static prohibition is necessary.

A.1.5 *Evaluation.* Non-virtual inheritance has the great advantage of presenting overhead only when one uses MI (provided that the implementation uses thunks). But repeated inheritance is a major semantic drawback. Sound semantics are possible by mixing virtual and non-virtual inheritance, but it is a matter of either hand-made optimization, or global analysis. Indeed, separate compilation cannot predict that B and C will not have a common subclass and where the `virtual` keyword must be used. Only a subsequent diagnosis, when defining D, is possible, or an automatic devirtualization analysis on the whole hierarchy (Section 6.4.1, page 41). From a programmer's standpoint, mixing virtual and non-virtual inheritance may be quite complicated and this even worsens with inheritance protections. It should be possible to restrict the use of NVI to DFI by forbidding the definition of D. But this would limit reusability. Moreover, when the language enforces rooted hierarchies, DFI reduces to SI. In principle, this sound usage could be extended to diamonds in ESO situations (Section 3.3, page 23), for instance when the diamond root is empty, or when B or C represent the first case of ESO. However, the actual C++ specifications would still require explicit null-casts.

A.1.6 *Criticizing* C++. Multiple inheritance in C++ has been commented and criticized [Cargill 1991; Waldo 1991; Sakkinen 1992], often for opposite reasons as either virtual or nonvirtual inheritance is blamed. Cargill [1991] criticizes virtual

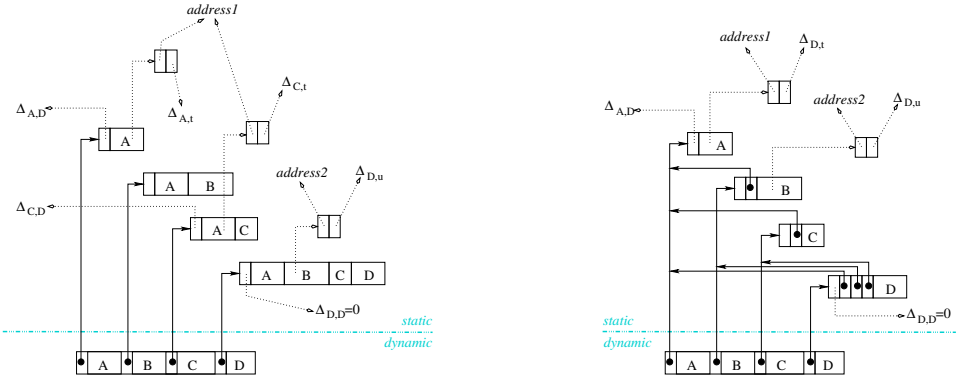


Fig. 13. Standard (left) and compact (right) method tables—dotted arrows represent containment

inheritance and it is significant that Waldo's answer [1991] is based on an example which needs multiple subtyping (MST, see Section 5) rather than full MI (see also Lippman's quotation, Section 3.3, page 24). The main criticism may be that the implementation must be understood, e.g. by reading [Lippman 1996], in order to understand the language. Specifications are not always clear and it is possible that, in complicated cases, actual compilers have different behaviours.

A.2 Compact Method Tables

Another drawback of standard MI implementation is its static space overhead, in the theoretical worst-case, but also in practice (see Table XII, page App–21). Two obstacles prohibit sharing tables of different τ_s for the same τ_d—i.e. shifts and variable method offsets. A solution may be to restore the symmetry between attributes and methods, by applying Invariant 3.2, which rules attributes, to methods—this solves the second point. The first point can then be reduced by two step shifting, as for attributes (5) and downcast (6), page 21. Therefore, the method table associated with a static type contains only methods introduced by the type, not the inherited ones. A method call must then begin with an upcast towards the type which introduces the method, as with attributes (5) when $T_p \neq \tau_s$. The cost of this technique is exactly the sum of the costs of upcasts and method calls, so the call sequence has 8 instructions and $4L + B + 2$ cycles. Instead of getting the second table from the cast object, one can use pointers between the different method tables (Figure 13, right). This reduces the cost to $3L + B + 1$ cycles because two `load`s can be run in parallel. In this case, the two step shift may indifferently go through either τ_d (6) or T_p (5). A minor overhead is that upcast structures are implemented twice, as shifts between subobjects and as pointers to method tables.

Evaluation. This technique reduces the static table size just at the cost of doubling upcast tables, together with 3 extra instructions for each method call such that $\tau_s \neq T_p$. The objective of reducing table size, hence static memory, might be hindered by this marked code enlargement. Only the first case of empty subobject optimization works, as merging now requires that both the subobject and its method table are empty. Moreover, time efficiency is reduced and the theoretical

L extra cycles likely underestimate the effective extra overhead. Anyway, it seems that standard implementation offers, in some way, the best tradeoff between time and static memory. Even in this compact variant, the worst case remains cubic, because constant-time upcasts require an encoding of the transitive closure of the inheritance relationship. Other variants exist, but none of them are better.

A.3 Less Indirections with VBPTRs

In contrast with the previous variant which attempts to reduce the static memory cost, some optimizations aim at reducing time overhead, at the expense of dynamic memory. A way to do this involves moving the upcast tables $\Delta^{\uparrow}_{\tau_s}$ from the method tables to the objects themselves. Then, shifts common to all instances can be replaced by pointers to subobjects, specific to each instance (Figure 14, page App–7, left). In C++ jargon [Ellis and Stroustrup 1990], they are called *virtual base pointers* (VBPTR) and, according to [Sweeney and Burke 2003], this technique is used in several C++ compilers. A similar optimization consists of replacing $\Delta_{\Downarrow}$ by a pointer to τ_d subobject in the object layout. Upcasts are viewed by standard implementation as methods—this imposes a method table access that increases cache-miss risks (see Section 3.2.1, page 21). VBPTRs can be viewed as immutable attributes that can be copied in each subobject since they are immutable. The general case of attribute access ($T_p \neq \tau_s$, page 21) reduces to:

```
load [object + #castOffset], object                                  2L
load [object + #attributeOffset], attribute
```

Evaluation in Standard Implementation. $\Delta^{\uparrow}_{\tau_s}$ tables are, in the worst case, quadratic in number and cubic in size. Therefore, with standard MI implementation, in each instance, the VBPTR number is quadratic in the number of superclasses. Moreover, this worst case is not theoretical, since it occurs with SI—a n class chain induces $n(n-1)/2$ VBPTRs. Splitting a class C (Fig. 2, page 14) adds overhead in all its subclasses D, which is linear in the number of classes between C and D. In contrast, assuming a uniform distribution of attributes introduced in each class, the number of attributes may be considered as linear in the number of superclasses. Thus, VBPTRs could easily occupy more space than attributes—this is indeed confirmed by our statistics (Table IX, page App–18). Instance creation is no longer linear but quadratic in the number of superclasses, since VBPTRs must be initialized. Furthermore, VBPTRs are compatible only with the first case of empty subobject merging—indeed, in the second case, the F subobject would not be empty since it must contain a VBPTR to the E subobject (Section 3.3, page 23). As for the pointer to τ_d, the overhead is smaller, as it is equal to the subobject number, but the gain is also smaller as shifts to the dynamic type are less frequent. Overall, the large dynamic overhead of VBPTRs is likely not counterbalanced by a small gain in time, mostly due to the reduction in cache misses.

Evaluation in NVI and optimizations. With pure NVI, upcasts are static and $\Delta^{\uparrow}$ tables useless. When mixing virtual and non-virtual inheritance, the VBPTR overhead is far lower than with SMI. The reason for this is twofold. First, with NVI, a class can reuse the VBPTRs of its direct non-virtual superclasses, which is not the case with $\Delta^{\uparrow}_{\tau_s}$ tables since they are relative to τ_s. Second, two kinds of VBPTRs can be distinguished: e-VBPTRs are *essential* because they reference superclasses

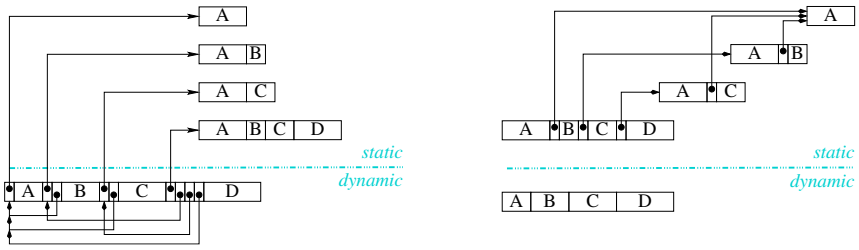


Fig. 14. Implementation variants with VBPTRs (left) and method table flow (right).

reachable only through $\prec_{\mathrm{v}}$, whereas i-VBPTRs are *inessential* and can be replaced by a static shift w.r.t. an e-VBPTR. Implementing only e-VBPTRs adds one extra shift to upcast, but no overhead for attribute access as the shift can be added to the attribute offset at compile-time. Furthermore, pointers to τ_d are required only in root (w.r.t. $\prec_{\mathrm{nv}}$) subobjects.

Sweeney and Burke [2003] propose a general framework for analyzing the space overhead of different variants, according to the distribution of data between class tables and object layout. [Gil and Sweeney 1999; Eckel and Gil 2000] present statistics on the VBPTR cost in some large benchmarks and they propose optimizations aimed at reducing VBPTR overhead. Some of them are global and compare badly with the invariant-reference techniques used in global compilation, like coloring (Section 6.3, page 39) or tree-based dispatch (Section 6.2.2, page 38). Other optimizations with bidirectional tables (Section 6.3.3, page 41) work in separate compilation. Overall, with an optimal balance of virtual and non-virtual inheritance, the dynamic overhead of VBPTRs remains high, but it is within the same order of magnitude as the number of subobjects (see Tables VII, page App–17, to IX).

A.4 Method Table Flow

A radical alternative to previous implementations involves moving links between object layout and method tables from data structures to program data flow. Any value is then twofold—the object itself and its method table—as if each variable, parameter and attribute were duplicated. Of course only polymorphic variables are concerned—monomorphic variables, e.g. typed by a primitive type, do not need tables. The object layout consists only of fields for attributes, which are almost doubled because of the tables but without any pointer to method tables or VBPTRs. Therefore, access to attributes should be made by simulating accessors. A secondary consequence is that method parameters and returned values are almost doubled.

Method tables have now the same structure as in SMI, and all method tables of the same τ_d are linked with each other by VBPTR-like pointers. SST reference Invariant 2.1, page 10, is respected for the object part of the value, whereas the SMI position Invariant 3.1, page 17, rules the method table part. All pointer adjustments needed by standard implementation, for upcast or receiver, are done on the method table in such a way that the method table part of a value is always the table corresponding to its static type (Figure 14, right).

In this approach, any variable `x` is twofold: `x` itself which references an object and the method table `tablex`. In the polymorphic assignment `x:=y`, the appropriate

table is obtained from `tabley`:

```
load [tabley + #deltaOffset], tablex                    L
move y, x
```

Method call to `x` follows:

```
load [tablex + #methodOffset], method                   L + B
call method
```

Evaluation. Maintaining a systematic flow of method tables in parallel to data flow seems at first glance to be a time-efficient solution, just at the cost of almost doubling memory space, both dynamic and static, in the code and in the data structures. However, the actual overhead might be less than this pessimistic view. Many data are monomorphic and this technique would have no influence on them. Dynamic memory overhead will be partially counterbalanced by the gain of all pointers, w.r.t. to method tables or VBPTRs. In fact, the proposed implementation may be as time efficient as standard implementation with VBPTRs, whereas VBPTRs and pointers to method tables often more than double the dynamic memory (see Table IX, page App–18).

Increasing the code size with extra parameters is partly counterbalanced by the fact that there is no longer a need for access to the object to get its method table—moreover, an optimized compiler will remove all accesses to tables which are not used. The advantages of the approach certainly depend on the way primitives types and object-oriented programming are integrated and on the proportion of object-oriented accesses to primitive operations.

Overall, while the approach is probably not an efficient alternative for a pure object-oriented language, it might be considered for some specific niches. Access to interface-typed entities in a MST framework is an example (Section 5, page 30). This may also be a good technique for implementing universal types, i.e. common superclasses of primitive types and standard classes, thus avoiding *boxing* as in JAVA (see Appendix B.1). We originally considered this technique as pure speculation, but it seems to be very close to the way method invocation is implemented in the EMERALD language, where method tables are generated at load-time in a lazy way [Jul 2008].

A.5 Conclusions on Subobject Variants

The main drawback of standard MI implementation is that its overhead does not depend on whether MI is used or not. NVI is an attempt at avoiding this drawback to the detriment of semantics. A general solution could be that languages allow the programmer to express the fact that a class should not be specialized in MI or, more generally, with non-diamond-free subclasses, but this would be to the detriment of reusability. The subobject-based implementation variants seem hopeless; compact method tables add time overhead and may be unable to effectively gain on static memory, whereas VBPTRs can be envisaged only in a context of massive NVI. Actually, the fact that some compilers implement VBPTRs indicates that their designers could not imagine programs that would rely only on virtual inheritance.

B. COMPLEMENTARY MECHANISMS

Most languages complement the core of object-oriented programming that we have examined with a set of features which may require specific implementation.

B.1 Polymorphic Use of Primitive Types

The object layout described here is common to all objects, instances of usual classes, apart from primitive values that are instances of primitive types, e.g. numbers and booleans. With static typing, there is usually no need to encode the primitive type in the value itself, as it must be done in dynamic typing, e.g. in LISP or SMALLTALK [Ingalls et al. 1997]. The case of higher-order types will be examined in Appendix B.2. When the receiver static type is primitive, the call is static. So the only issue with primitive types regards their polymorphic usage, when the language has either abstract primitive types (e.g. `Number`) or some *universal types* which are supertypes of both primitive types and usual classes, like `Any` in EIFFEL and `Object` in C# and JAVA 1.5. A value of such a universal type may thus be a primitive value or a usual object and the implementation must address this exception to reference Invariant which is close to pointer adjustment with subobjects.

The specific case of `self` can be solved by *customization* (Section 6.4.4, page 42). Its incompatibility with separate compilation should not be a problem since primitive types are usually defined in the language kernel. The methods defined in `Any` and `Number` can thus be copied in their primitive subtypes. For other usages, there are two common solutions: *boxing* and *tagging*. Method table flow would likely represent an original alternative (see Appendix A.4).

Boxing and Unboxing. Universal types may assume a common layout with a pointer to method table. Hence, an upcast of a primitive value involves *boxing* it into a data structure containing the value and equipped with a method table corresponding to the primitive type. Conversely, besides the dynamic type checking, downcasts amount to *unboxing* the value. Boxing and unboxing would be burdensome for the programmer but automatic boxing is costly, even though data flow analysis can spare some boxes. Testing the physical equality of two universally typed entities is also complicated, since programmers are presumed to be interested in the equality of boxed values, not in the equality of boxes. Therefore, when x and y are boxes, $x = y$ either if x and y are the same address, or if x and y have the same method table (i.e. same type), the same value in their first field, and if the common method table contains a certificate ensuring that it is a primitive type. When x and y are, respectively, a primitive value and a box, $x = y$ if the method table of y is that of the box-type of x, and if y has the value of x in its first field. Note that this cannot be done with a user-defined method like `equals` in JAVA, because such a method is specified for testing logical rather than physical equality.

Tagging. An alternative to boxing currently used with dynamic typing is *tagging*. Each value is represented by a machine word, e.g. 32 bits. References are aligned addresses, i.e. multiples of 4. Hence, a primitive value can be represented by an odd unsigned integer. A subset of the word (e.g. a byte) can now be reserved as a tag for the value type. When the static type of the receiver is some universal type, e.g. `Any` or `Number`, method invocation is implemented by *type prediction*

(Section 4.3, page 28), with a BTD that tests whether the value is tagged and checks the tag against all expected type tags. When the value is a reference, the usual implementation is used. Of course, encoding the values on less than 32 bits may seem restrictive, and could be reserved for ubiquitous types like booleans, characters. Regarding integers, 24-bit integers are a severe limitation, but tagging might be coupled with unlimited-magnitude integers (`bignums`).

B.2 Genericity

Parametrized classes can be thought of as metaclasses that are instantiated by substituting types or classes for their formal types. This results in ordinary classes that can be compiled and implemented as such. This textual substitution, called *heterogeneous* implementation by Odersky and Wadler [1997], is used for C++ templates. It currently has a major drawback, namely that the parametrized class itself is not compiled, not even type-checked, although all arguments in favour of separate compilation apply here—see for instance the critique of C++ templates by Lippman [1996]. In contrast, an actual compilation of the parametrized class allows the same code to be shared between its different future instantiations. This is the JAVA approach, that Odersky and Wadler call *homogeneous*. Both approaches have pros and cons. When instantiated with primitive types, heterogeneity produces much more efficient code than homogeneity, which yields omnipresent boxing and unboxing. Moreover, heterogeneity allows the programmer to consider formal types as usual types. It is, for instance, easy to implement mixins as *parametrized heir classes* [VanHilst and Notkin 1996; Smaragdakis and Batory 2002]—see Section 5.4, page 34. In contrast, homogeneity involves *type erasure*, by replacing the formal type by its bound. It follows that a formal type cannot create instances and different instances of the parametrized class cannot distinguish their own instantiation types, e.g. a `List`⟨`Integer`⟩ cannot be distinguished at runtime from a `List`⟨`Object`⟩. Obviously, an efficient implementation should involve a mixing of heterogeneity, for primitive types, and homogeneity, for all other types, both combined with runtime encoding of instantiation types. This is basically the .NET solution [Kennedy and Syme 2001], but it still remains a matter of current research on other platforms.

B.3 Type Variant Overriding

Up to now, we assumed that method signatures were invariant. However, in a type safe context, method overriding is ruled by the *contravariance rule*—the parameter type must be contravariant and the return type covariant [Castagna 1997]. Actually, some authors argue that strictly contravariant parameters are not needed, since the models that programmers want to implement are mostly covariant. See for example [Cook 1989; Weber 1992; Castagna 1997] for a critique of covariance and [Meyer 1997; Shang 1996] for a defence. Anyway, implementing variant overriding comes into question, either for type-safe overriding—especially for covariant return type as in C++—for pure covariant languages like EIFFEL, for *virtual types* [Torgersen 1998], or for some specific unsafe features like JAVA arrays. Incidentally, the EIFFEL requirement for global compilation is due to its unrestricted covariance—with separate compilation, EIFFEL would be both unsafe and inefficient.

With reference-invariant implementations, type-safe overriding does not require any specific implementation, apart from primitive values (Appendix B.1). With

subobject-based implementations, any type overriding requires pointer adjustment. For the return type, the best solution is likely an adjustment in the thunk, *after* the callee has returned a value, thus implying a second function call. Alternatively, it could be done in the caller as a type-safe downcast. For parameter types, this is less costly, since the adjustment can be done *before* jumping to the callee. Covariant parameter types can be further improved by allocating a table entry for each method signature, as with static overloading. Each method entry contains a thunk which performs the subtype tests required for its specific signature.

B.4 Multiple Dispatch

With multiple dispatch, method selection depends on the dynamic type of several or even all parameters. CLOS [Steele 1990] specifies it under the form of *generic functions* which are usual LISP functions dispatched between all *methods* of the considered function. Generic functions are orthogonal to classes instead of being included in them as methods usually do. in this form, multiple dispatch can be understood as single dispatch in the Cartesian product of class sets. In this framework, dynamic lookup strategies [Chen et al. 1994] and cache-based techniques [Kiczales and Rodriguez 1990] have been examined. Authors' arguments for these non-constant-time approaches are that all experiments show that multiple dispatch is rarely used, i.e. about 95% of generic functions dispatch on a single parameter, and 95% of the remainder dispatch only on two parameters. [Dujardin et al. 1998] propose an implementation, based on a generalization of the coloring principle, with several indirections, that is time-linear in the number of parameters. [Holst et al. 1998; Zibin and Gil 2002] propose alternatives. However, all of these approaches are in dynamic typing.

Generic functions are not adapted to the modular view of classes for which methods are encapsulated in classes. Some proposals have been made which are both modular and statically typed [Mugridge et al. 1991; Boyland and Castagna 1996; Castagna 1997]. We do not take their possible differences into account. Usual methods defined in a class are replaced by *multi-methods*, which involve several *branches* that differ from each other by their parameter types. As usual, the terminology is not uniform; [Castagna 1997] uses the term *overloaded functions* (which must not be confused with *static overloading*) to name what we call here *multi-methods*, whereas [Mugridge et al. 1991] uses *multi-method* instead of *generic function*, with the same meaning as in CLOS ancestors (COMMON LOOPS, NEW FLAVORS). In our terminology (Section 1.2, page 4), *method* stands for a *generic property*—SMALLTALK *method selector* and CLOS *generic function* are analogues—for which several definitions exist in different classes. A *multi-method* is then a method definition composed of several *branches*. The role of dispatch is now to select a single branch. Hence, dispatch has two steps: (i) selection of a multi-method, based on the receiver's dynamic type; and (ii) selection of the branch in the selected multi-method, based on the dynamic type of all remaining parameters. Typing rules statically ensure that, for all actual dynamic types, a single more specific branch exists [Castagna 1997]. Implementation of step (i) is as usual. The selected code, i.e. the multi-method, must dispatch between different branches according to the dynamic types, in a way that is very similar to a `typecase` construct. Some kind of tree-based dispatch might be a solution. If this tree is generated at compile time, it must be

based on subtype tests, whereas global link-time generation may use BTD. The same optimization as for covariant parameter types can save some tests. Each branch signature then has its own entry in the method tables, and static selection dispatches between the different signatures, whereas each table entry contains the dynamic dispatch required for its specific signature. Overall, multi-methods can be implemented in such a way that overhead occurs only when multiple dispatch is actually needed at run time.

B.5 Calling `super`

All languages offer a mechanism for calling the overridden method from the overriding one. In SMALLTALK and JAVA, this is performed with the keyword `super`, a pseudo-variable like `self`, which consists of calling the superclass method on the current receiver, i.e. `self`. In SI, the superclass method is unambiguously determined, and the call is static. In multiple inheritance, *the* superclass method is not uniquely determined in the general case, and the mechanism has various specifications. From the implementation standpoint, there are only two variants. Besides the syntactic differences, C++ (with ::) and EIFFEL (with `Precursor`) involve a static call, like `super` in SI. In contrast, CLOS `call-next-method` [Steele 1990] and SCALA `super` [Odersky et al. 2008] consist of calling the next method in the so-called *linearization* of the superclasses of the receiver's dynamic type. The next method now depends on the dynamic type. In the diamond example in Figure 11, page App–2, the linearization of D (resp. B) may be $\{D, B, C, A\}$ (resp. $\{B, A\}$); so, in B, the next method will be in C (resp. A). A simple solution, used in SCALA, involves assigning an extra offset in the method tables, as if each `call-next-method` introduce a new method. Interested readers are referred to [Ducournau et al. 1994; Ducournau and Privat 2008] for an in-depth discussion on linearizations and calls to `super`.

B.6 `Null` Pointers

Any variable or attribute typed by a class must be explicitly initialized before any sound access. An almost universal implementation relies on a distinguished `null` value that serves as a default initial value and the receiver is tested against it before each access. It yields ubiquitous `null`-checks that are more boring than costly since they are usually well predicted. Data flow analysis may detect initialized and uninitialized local variables, but this is more difficult for attributes, which may be initialized some time after instance creation. A partial solution would be a type system that distinguishes between types according to whether the value can be `null` or not [Fähndrich and Leino 2003]. This is, however, the focus of current research.

An alternative might be a `null` object whose all method implementations would signal an exception. This would work well with method invocation but not with attribute access. This instance should be allocated in the code area, which is assumed to be read-only, in order to prevent any assignment. However, read accesses are possible, thus propagating `null` values, which may make debugging difficult. This is actually a current issue with the usual `null`-check. Another alternative depends on the hardware and operating system—for instance, in [Alpern et al. 1999], `null` has address 0, and all offsets are negative as, in the AIX system, negative addresses raise an interrupt. A similar solution is likely used in many compilers.

B.7 Garbage Collection

Automatic memory management is a reliability argument that has been promoted by successive designers of LISP, SMALLTALK, EIFFEL and JAVA. The discussion of the various *garbage collection* (GC) techniques is far beyond the scope of this article—interested readers are referred to the renowned surveys of [Wilson 1992; Jones and Lins 1996]. However, our survey concerns garbage collection to some extent. Two examples will illustrate this point. In usual JAVA implementations, classes precisely describe their instances, and methods their stack blocks, in such a way that the garbage collector always knows if a machine word is a reference—that always points to an object at offset 0—or not. The garbage collector is said to be *type accurate*. In C++, due to its implementation and compatibility with C, there is no way for the garbage collector to be sure that a machine word is a reference or not; hence, it must be *conservative*, as Boehm's [1993] commonly used library. Moreover, with subobjects, a reference can point to an object at any position inside it. Therefore, subobject-based implementations must yield marked overhead for garbage collection, whereas unidirectional invariant-reference object layout should allow for more efficient memory management. Overall, memory management and object representation are closely related, and a marked gain should be obtained when both are designed in relation to each other. For instance, Gagnon and Hendren [2001] propose a bidirectional layout designed especially for optimizing garbage collection, by assigning negative offsets to references and positive offsets to primitive values. With bidirectional coloring (Section 6.3.3, page 41), the adaptation to type accurate collectors likely involves duplicating the pointer to the method table when the object layout has negative offsets, in such a way that the first word of the object layout always points to a data structure that describes the object layout. In contrast, using an unoptimized collector can yield marked overhead [Ducournau et al. 2009].

C. MEMORY OCCUPATION BENCHMARKS

Some large benchmarks have been used in the object-oriented implementation community, for instance by [Driesen and Hölzle 1995; Vitek et al. 1997; Eckel and Gil 2000; Zibin and Gil 2002], for an assessment of various techniques and algorithms. Many people contributed to these benchmarks. They are accessible on the author's website `http://www.lirmm.fr/~ducour/Benchmarks`. These benchmarks consist of abstract class hierarchies and they can only support measurement of the size of the various data structures that are needed by object-oriented implementations. Therefore, the statistics presented here can only serve as an assessment of the memory occupation required by the tested implementations.

C.1 Tested Techniques

Seven techniques have been considered:

—ideal single-subtyping (SST, Section 2, page 9) that provides rough parameters on the benchmarks and a reference for all other techniques, even though SST is not directly applicable as all benchmarks are in unconstrained MI;

—standard MI (SMI, Section 3, page 17), that represents the usual C++ implementation of virtual inheritance;

—its variant with the empty subobject optimization (ESO, Section 3.3, page 23), that is presumed to be the best subobject based MI implementation under the OWA; in some cases, the effect of static merging (ESO_1) has been isolated;

—pure non-virtual inheritance (NVI, Appendix A.1, page App–1) that would yield repeated inheritance on our benchmarks with unrestricted MI;

—ideal devirtualization (DVI, Section 6.4.1, page 41), that may be understood as the best subobject-based implementation, with a sound multiple inheritance semantics, under the CWA;

—coloring, as well unidirectional (COL_1) as bidirectional (COL_2), that is presumed to be the best technique without global optimizations under partial CWA (Section 6.3, page 39).

Finally, no simulations of tree-based techniques (Section 6.2.2, page 38) have been performed as their measurement would require programs, while our benchmarks are libraries. Regarding multiple-subtyping (MST) implementations, no specific measures are presented here. Finally, interested readers are referred to [Ducournau 2008; Ducournau and Morandat 2009] for in-depth experiments on perfect hashing.

C.2 Benchmark Description, Interpretation and Correction

Each benchmark represents a class hierarchy and consists of a file of class descriptions that are akin to the *external schemata* required for separate compilation (Section 1.3, page 6). Each class description consists of four items: the class name, the list of its direct superclasses, and the two lists of attributes and methods *defined* in the class. They have been produced and used mostly for assessing techniques for subtyping test and method call, so they do not often include information about attributes. Therefore, two groups of benchmarks can be distinguished. The first one includes data on attributes: Unidraw is a C++ program mainly in SI; Lov and Geode are Eiffel-like programs thta makes intensive use of MI; SmartEiffel is the GNU Eiffel compiler (Section 6.4.7, page 43); PRMcl is the Prm compiler-linker (Section 6.5.2, page 44); finally Java1.6 is the only Java benchmark that includes data on attributes. The second group involves several Java benchmarks (from IBM-SF to HotJava) plus some benchmarks of different languages which are dynamically typed but included here only for comparison (Cecil, Dylan, Harlequin).

The content of the benchmarks is also questionable. One should expect that only pure object-oriented data are included: typically, `static` methods and variables, or non-`virtual` methods should be excluded. We can only certify that non-object-oriented data has been removed from the SmartEiffel, PRMcl and Java1.6 benchmarks[2]. These benchmarks come from various languages whose specifications vary, especially for all that concern name interpretation, e.g. overloading, multiple inheritance. Static overloading has been dealt with by concatenating parameter types to method names. Regarding multiple inheritance, we have considered that there was no *introduction overloading* (see Section 2.1, page 11), i.e. each attribute and method is introduced by a single class. For some benchmarks, the actual language specifications might differ, but the effect on the statistics should not be significant.

[2]Thanks to Floréal Morandat and Olivier Zendra.

Table V. Class and edge numbers

	classes								edges	
		ESO_1	ESO	DVI			COL			DVI
name	total	empty		virtual	e+v	double	core	leaves	total	virtual
Java1.6	5075	1298	1535	53	136	239	1422	3825	6363	168
Geode	1318	163	436	78	106	231	989	732	2486	204
Unidraw	614	115	81	3	1	6	25	481	623	4
Lov-obj-ed	436	39	132	21	40	75	271	218	747	61
SmartEiffel	397	33	116	3	9	12	67	311	437	11
PRMcl	479	38	209	16	14	37	133	294	541	28
Total	8319	1686	2509	174	306	600	2907	5861	11197	476
IBM-SF	8793			346		524	4770	6001	12033	458
JDK1.3.1	7401			63		299	1512	5806	8605	227
Orbix	2716			10		31	271	2440	2923	26
Corba	1699			35		44	383	1473	2072	52
Orbacus	1379			25		64	502	954	1812	52
HotJava	736			3		55	217	525	927	33
Cecil	932			31		76	306	601	1127	50
Dylan	925			8		13	65	806	953	11
Harlequin	666			24		84	278	371	907	59
Total	25247			545		1190	8304	18977	31359	968

The table presents, from left to right, the numbers of: (1) all classes, (2-3) two kinds of empty classes, without attributes, (4) virtual and (5) virtual empty classes, (6) double classes, (7) core and (8) leaf classes; then the number of all direct edges ($\prec_d$) and virtual edges ($\prec_v$).

Moreover, these benchmarks are often libraries, not single applications. It is thus difficult to extrapolate from them the size appropriate for typical applications and to determine the maximal number of classes, methods, attributes, etc. of applications. Anyway, some hundreds of classes seem common in object-oriented programs, e.g. the SmartEiffel and PRM compilers which represent programs, not only libraries. Palacz and Vitek [2003] report that their experiments did not exceed one thousand classes loaded from almost 10-thousand class libraries.

Finally, the statistics may differ from previous ones in several ways. The benchmark interpretation may be different—for instance, the interpretation of introduction overloading would be quite different with dynamic typing. Second, all methods are considered, whereas Yoav Zibin's benchmarks remove *degenerate methods*, i.e. methods which have only one definition. However, these degenerate methods might have been removed from some original JAVA benchmarks—this would explain the very low number of monomorphic methods in Table XI (see also Note 3, page App–20). Furthermore, some measures are uniquely determinate (class and method numbers) whereas many other (ESO, DVI, COL and even NVI) are the result of either an arbitrary choice or an approximate optimization. There is thus very little chance that two different experiments, by two different programmers, will give the same number. NVI, DVI and ESO depend on the choice of the direct superclass whose layout is extended by the current class. In all three cases, our heuristics choose a class with the greatest number of methods, which tends to minimize the table size. Nevertheless, besides all of these small variations, the statistics presented hereafter provide a reliable indication of the relative cost of the measured techniques.

C.3 Class Hierarchies

Definitions. A *class hierarchy* is a partially ordered set (aka *poset*) of classes, denoted $(X, \prec)$. $(X, \prec_{\mathrm{d}})$, the transitive reduction of $(X, \prec)$, is a directed acyclic graph (aka *dag*). Several techniques require a more technical view of the class hierarchy. The *core* of a class hierarchy is the subset $X_c \subseteq X$ of classes which are concerned with MI, i.e. which have at least two direct superclasses, or have a subclass with two direct superclasses. X_c is the minimal set such that

$$C \prec_{\mathrm{d}} C_1 \ \&\ C \prec_{\mathrm{d}} C_2 \ \&\ C_1 \neq C_2 \Rightarrow C \in X_c$$
$$D \in X_c \ \&\ D \prec C \Rightarrow C \in X_c$$

Coloring, devirtualization and double compilation are based on the *conflict graph* (X_c, E_c) [Ducournau 2006], where

$$E_c = \{(C_1, C_2) \mid \exists D \in X_c, D \prec C_1 \ \&\ D \prec C_2 \ \&\ C_1 \not\preceq C_2 \ \&\ C_2 \not\preceq C_1\}.$$

Double classes X_d are the classes which must be used in the less efficient form in the double compilation approach, i.e. which are not a primary superclass of all their subclasses (Section 5.4, page 34); they form a minimal *vertex cover* of the conflict graph. A class has at most one direct superclass in $X \backslash X_d$. In the DVI approach, *virtual classes* are core classes which are inherited by some subclass through two disjoint paths:

$$X_v = \{C \in X_c \mid \exists (C_1, C_2) \in E_c, C_1 \prec_d C \ \&\ C_2 \prec_d C\}$$

C, C_1, C_2 and D (from E_c definition) form a diamond. *Virtual edges* are a subset $\prec_{\mathrm{v}}$ of $\prec_{\mathrm{d}}$, which is computed for devirtualization as a minimal subset of $X_d \times X_v$, the set of edges from a *double* class to a *virtual* one, such that $(X_c, \prec_{\mathrm{nv}}\}$ is acyclic, in the undirected sense (where $\prec_{\mathrm{nv}}$ stands for $\prec_{\mathrm{d}} \setminus \prec_{\mathrm{v}}$).

In the diamond example shown in Figure 11, page App–2, $E_c = \{(B, C)\}$, $X_v = \{A\}$, and $X_d = \{B\}$ (or $\{C\}$).

All of these structures (i.e. core, double and virtual class sets) are empty in a SI hierarchy and their size is a strong indication of the way multiple inheritance is used.

Class Numbers. Table V presents the cardinality of all of these sets. The number of classes without attributes is surprisingly high in all benchmarks, i.e. between 30 and 55%. There are also very few virtual classes and edges. Both observations mean that ESO and DVI can be very effective optimizations, though combining them would be useless since the number of classes without attribute or non-virtual direct superclass (column "e+v") is quite small. The number of double classes is roughly 25% of the core size. Both numbers are a good indication on how much MI is used.

In Java benchmarks, no specific measure of interfaces has been made. However, $X_d \cup X_v$ is a good approximation of the set of interfaces: a Java class is never virtual (except `Object`, as far as it can be considered as a class) and all classes might be in $X \backslash X_d$. Therefore, the set of interfaces is a solution to $X_d \cup X_v$, but it is likely that this heuristics misses some interfaces and replaces them by classes. In Fig. 7, page 30, $X_v = \{I, K, L\}$ and X_d might be $\{J, L, M\}$ or $\{K, A, B\}$, or some

Table VI. Statistics on attribute definition (a_C, A_C)

	introduced (a_C)				inherited (A_C)		
name	total	average		max	total	avg	max
Java1.6	8201	1.6	3.7	55	27580	5.4	137
Geode	2919	2.2	4.1	182	14392	10.9	217
Unidraw	1574	2.6	3.8	36	5103	8.3	47
Lov-obj-ed	1262	2.9	4.8	74	3554	8.2	105
SmartEiffel	977	2.5	3.9	39	1956	4.9	44
PRMcl	578	1.2	2.5	28	2411	5.0	29
Total	15511	1.9	3.8	182	54996	6.6	217

The table presents, from left to right: the total number of attributes, with the average and maximum number of introductions per class and non-empty class (a_C), then the same statistics for inherited attributes (A_C). The total number of inherited attributes represents the total space occupied by one instance per class (in machine words or table entries). The "Total" line represents the sum of all totals, the weighted mean of all averages, and the maximum of all maximums.

Table VII. Absolute overhead in object layout (n_C, O_C vs A_C)

name	A_C		NVI		DVI		ESO		ESO_1		SMI (n_C)	COL_1		COL_2	
Java1.6	5.4	137	1.6	18	3.1	20	2.2	19	4.2	22	5.4	23	1.0	1	1.0	1
Geode	10.9	217	10.5	140	8.3	28	8.5	33	13.4	48	14.0	50	2.9	60	2.4	22
Unidraw	8.3	47	1.0	4	1.4	3	3.1	7	3.5	8	4.0	10	1.0	3	1.4	2
Lov-obj-ed	8.2	105	4.2	19	5.1	16	5.4	17	7.8	20	8.5	24	2.7	30	1.6	4
SmartEiffel	4.9	44	2.1	8	2.1	4	2.4	5	6.1	11	8.6	14	1.0	1	1.0	1
PRMcl	5.0	29	1.5	7	1.8	5	3.0	9	4.4	12	4.6	12	1.2	9	1.3	5
Total	6.6	217	3.1	140	3.8	28	3.5	33	5.9	48	6.9	50	1.4	60	1.3	22
IBM-SF			4.0	39	3.6	14					9.2	30				
JDK1.3.1			1.5	21	1.9	21					4.4	24				
Orbix			1.2	6	1.3	6					2.8	13				
Corba			1.7	19	1.9	11					3.9	18				
Orbacus			2.0	11	2.3	11					4.5	19				
HotJava			2.0	15	2.6	15					5.1	23				
Cecil			2.2	30	2.8	9					6.5	23				
Dylan			1.2	6	1.7	4					5.5	13				
Harlequin			2.2	15	2.7	16					6.7	31				
Total			2.4	39	2.6	21					6.1	31				

The table recalls the average and maximum attribute number (A_C) and presents the average and maximum numbers of compiler-generated fields, i.e. pointer to method table or holes, in the object layout (O_C), hence the absolute overhead of the different techniques. The SMI numbers are those of the superclass number (n_C).

combination of the two. In a complementary way, empty classes provide an upper bound to interfaces.

C.4 Object Layout and Dynamic Space

Table VI presents statistics on the number of attributes, i.e. introduced in a class (a_C) or the total number for the class ($A_C = \sum_{C \preceq D} a_D$). It is worth noting that the average number of attributes per class is quite uniform. The attribute number A_C represents useful information that must be compared with the number O_C of compiler-generated fields in the object layout. These fields are pointers to method tables or coloring holes (Table VII). Table VIII presents the relative overhead per object for the same data, i.e. the statistics of $(O_C - 1)/(1 + A_C)$, with the assumption that all implementations require at least one field.

Table VIII. Relative overhead in object layout ($(O_C - 1)/(1 + A_C)$)—average and maximum

name	NVI		DVI		ESO		ESO_1		SMI		COL_1		COL_2	
Java1.6	0.10	5.7	0.33	5.7	0.19	6.0	0.50	7.0	0.68	7.3	0.00	0.0	0.00	0.0
Geode	0.79	26.5	0.61	7.5	0.63	6.0	1.04	8.0	1.09	10.0	0.18	19.7	0.13	7.0
Unidraw	0.00	0.3	0.04	0.3	0.23	0.8	0.27	2.0	0.32	2.0	0.00	0.5	0.05	1.0
Lov-obj-ed	0.35	7.5	0.44	3.0	0.48	4.5	0.74	8.5	0.82	10.5	0.21	6.0	0.07	3.0
SmartEiffel	0.19	1.5	0.19	1.5	0.24	1.0	0.86	3.5	1.29	5.0	0.00	0.0	0.00	0.0
PRMcl	0.08	1.2	0.13	1.0	0.32	1.5	0.56	3.0	0.60	3.0	0.03	2.0	0.06	1.3

Table IX. Upcast tables (U_θ) and VBPTRs

name	e-VBPTRs			i-VBPTRs			DVI			$ESO=ESO_1$			SMI		
Java1.6	18080	4.	54	18367	4.	54	18501	4.	54	39535	8.	77	65816	13.	108
Geode	18510	14.	96	26477	20.	185	28240	21.	185	89547	68.	481	94992	72.	490
Unidraw	244	0.	3	264	0.	7	264	0.	7	3310	5.	28	4711	8.	45
Lov-obj-ed	2768	6.	32	3170	7.	40	3310	8.	41	10111	23.	117	11503	26.	118
SmartEiffel	446	1.	3	2102	5.	21	2102	5.	21	6678	17.	54	13732	35.	89
PRMcl	522	1.	10	692	1.	13	694	1.	13	4081	9.	53	4558	10.	53
Total	40570	5.	96	51072	6.	185	53111	6.	185	153262	18.	481	195312	23.	490
IBM-SF	48083	5.	28	66398	8.	53	68547	8.	54				374677	43.	302
JDK1.3.1	8371	1.	29	8676	1.	39	8840	1.	39				64498	9.	87
Orbix	1230	0.	7	1246	0.	9	1378	1.	10				8678	3.	51
Corba	2448	1.	25	2795	2.	32	3045	2.	34				11846	7.	95
Orbacus	2513	2.	13	2553	2.	13	2824	2.	13				12519	9.	73
HotJava	1434	2.	17	1434	2.	17	1457	2.	17				8729	12.	74
Cecil	3192	3.	35	4096	4.	54	4133	4.	54				18357	20.	183
Dylan	668	1.	4	1693	2.	13	1693	2.	13				12434	13.	63
Harlequin	1861	3.	31	3332	5.	50	3343	5.	50				14317	21.	176
Total	69800	3.	35	92223	4.	54	95260	4.	54				526055	21.	302

The table presents the total, average and maximum number of fields of upcast structures, either in the object layout with VBPTRs in the DVI case, or in the method tables. These upcast tables represent the U_θ parameter that is used in the static size evaluation (Section C.5, page App–22).

Overhead in Object Layout. With SMI, O_C is the number n_C of superclasses of C, including C itself—hence, it also gives a good indication on how much specialization is used. On most benchmarks, there are, on average, more pointers to method tables than attributes. The relative overhead (Table VIII) is even more impressive—it shows that, in the worst case, the number of compiler-generated fields can be 10-fold the attribute number. ESO_1 is a small improvement, but ESO and DVI reduce the overhead by 40%. One should not be surprised by the fact that, in the case of Geode, NVI is worse than ESO and DVI (on average) and even worse than SMI (in the worst case)—indeed, the worst-case complexity of NVI is exponential.

With coloring, the hole number is far lower. Note that, with bidirectional coloring (COL_2), O_C includes the extra pointer to the method table required for efficient garbage collection, when the layout of C involves negative offsets (Sections 6.3.3, page 41 and Appendix B.7, page App–13). However, the worst-case relative overhead remains high with unidirectional coloring, and this must be due to the fact that heuristics were not designed to minimize this parameter, but only the overall overhead. Anyway, this is a strong argument in favor of bidirectional coloring and it might be an argument for considering accessor simulation.

Table X. Statistics on method introduction and definition (m_C, $m_C^{\mathtt{def}}$, M_C)

name	introduced (m_C)			defined ($m_C^{\mathtt{def}}$)			inherited (M_C)			
Java1.6	22121	4.4	286	35352	7.0	291	190289	37.5	670	0.2
Geode	8078	6.1	193	14214	10.8	207	305560	231.8	880	2.9
Unidraw	1752	2.9	103	3328	5.4	103	14781	24.1	124	1.4
Lov-obj-ed	3631	8.3	117	5026	11.5	127	37436	85.9	289	2.4
SmartEiffel	4854	12.2	222	7865	19.8	222	53704	135.3	324	2.8
PRMcl	2369	4.9	115	3793	7.9	115	37517	78.3	208	3.3
Total	42805	5.1	286	69578	8.4	291	639287	76.8	880	0.2
IBM-SF	25000	2.8	257	116152	13.2	320	394375	44.9	346	0.2
JDK1.3.1	9567	1.3	149	28683	3.9	150	142445	19.2	243	0.2
Orbix	1135	0.4	64	3704	1.4	78	22637	8.3	109	0.7
Corba	627	0.4	43	3201	1.9	50	13578	8.0	67	1.3
Orbacus	1716	1.2	74	4996	3.6	79	24877	18.0	137	1.1
HotJava	1310	1.8	80	3397	4.6	85	25149	34.2	189	2.6
Cecil	2743	2.9	61	4208	4.5	62	73366	78.7	156	2.9
Dylan	814	0.9	64	1784	1.9	64	71308	77.1	139	9.5
Harlequin	416	0.6	62	1016	1.5	67	23167	34.8	129	8.4
Total	43328	1.7	257	167141	6.6	320	790902	31.3	346	0.1

The table presents the total, average and maximum numbers of methods, according to whether they are introduced, defined (i.e. implemented) or inherited in a class. The last column represents the percentage of occupied entries in the large class-method dispatch matrix (Section 6.2.2, page 38).

Upcast Tables and VBPTRs. Table IX shows the space occupied (in words) by the structures required for upcast in subobject-based implementations. Unsurprisingly, the numbers for SMI are much larger than the numbers of compiler-generated fields (Table VII)—this confirms that VBPTRs (Appendix A.3, page App–6) cannot be a solution for virtual inheritance. ESO improves these numbers, but only slightly because only the first case of merging saves on upcast structures. In contrast, DVI is a great improvement—upcast tables are 3 times smaller in the worst case, and VBPTRs are still fewer. On average, the difference between (essential) e-VBPTRs and (inessential) i-VBPTRs is not significant. However, the overhead of DVI with e-VBPTRs is far greater than that of attribute coloring.

Conclusion on Object Layout. These statistics clearly show that (i) the coloring overhead is very low, (ii) that of the usual implementation of C++ virtual inheritance is very high, even though (iii) empty-subobject optimization would be a great improvement; finally (iv) VBPTRs are unreasonable with virtual inheritance.

C.5 Method Tables and Static Space

Notations and Method Numbers. In the following, m_C (resp. $m_C^{\mathtt{def}}$) is the number of methods *introduced* (resp. *defined*) in class C, and M_C is the total number of methods defined in, or inherited by C: $M_C = \sum_{C \preceq D} m_D$.

Table X presents statistics on these three parameters m_C, $m_C^{\mathtt{def}}$ and M_C. Note first that, contrary to attributes, method numbers vary between benchmarks, with a ratio greater than 10 for introduction and definition, and up to 70 for inherited methods[3]. Moreover, from Tables VI and X, the number of attributes (A_C) and methods (M_C) in a class is clearly far from 2^{15}; thus, an implementation of Δs and

[3]JAVA benchmarks present an exception to these observations. However, the method numbers in Tables X and XI rise a question about these benchmarks. The Java1.6 benchmark has been

Table XI. Statistics on monomorphic and polymorphic methods

name	mono		poly	defined		
Java1.6	18443	0.83	3678	16909	4.6	433
Geode	7294	0.90	784	6920	8.8	773
Unidraw	1419	0.81	333	1909	5.7	42
Lov-obj-ed	3219	0.89	412	1807	4.4	225
SmartEiffel	4450	0.92	404	3415	8.5	53
PRMcl	1965	0.83	404	1828	4.5	119
Total	36790	0.86	6015	32788	5.5	773
IBM-SF	5344	0.21	19656	110808	5.6	2245
JDK1.3.1	3889	0.41	5678	24794	4.4	548
Orbix	350	0.31	785	3354	4.3	351
Corba	321	0.51	306	2880	9.4	445
Orbacus	357	0.21	1359	4639	3.4	143
HotJava	582	0.44	728	2815	3.9	72
Cecil	2217	0.81	526	1991	3.8	374
Dylan	556	0.68	258	1228	4.8	69
Harlequin	275	0.66	141	741	5.3	50
Total	13891	0.32	29437	153250	5.2	2245

The table presents the number and rate of monomorphic methods, that have a single definition, then statistics of the polymorphic methods, with their total, average and maximum number of definitions.

offsets with short integers is valid. This explains why, in the following, the numbers for upcast tables and shifts will be divided by 2.

Table XI presents the statistics of monomorphic methods—i.e. methods that have single definitions—together with the statistics of definition numbers for polymorphic methods. The ratio of monomorphic methods is very high; this confirms an observation often reported in the literature[3].

These statistics show that calls to a large majority of methods could be compiled as static calls and a simple *class hierarchy analysis* (CHA) would be a great improvement if it were applied at link-time (Section 6.5, page 43). They also explain why the assumptions made by adaptive compilers are generally not contradicted by further loadings (Section 6.6, page 45). In contrast, some of the few polymorphic methods may have up to 2245 method definitions. So binary tree dispatch (BTD, Section 6.2.2, page 38) might be defeated in some specific situations and combining it with coloring is certainly better.

Method Tables. According to the different techniques θ in {SST, DVI, NVI, ESO, SMI}, the total size of method tables, denoted T_θ, can be computed by the following formulas:

$$T_\theta = \sum_C f_C^\theta, \text{ where } f_C^\theta = \begin{cases} M_C = \sum_{C \preceq D} m_D & \theta = \text{SST} \\ \sum_{C \preceq D} M_D & \theta = \text{SMI} \\ \sum_{C \preceq_{\text{eso}} D} M_D & \theta = \text{ESO} \end{cases} \tag{11}$$

extracted independently from other JAVA benchmarks and it presents method numbers that are markedly higher, especially for monomorphic methods—this must be due to the fact that degenerate methods have been removed from other JAVA benchmarks. In spite of this removal, there are many remaining monomorphic methods because degenerate methods only consider the method name, not the introduction class.

Table XII. Statistics on method table size (T_θ)

	size in K-entries						ratio to SST				
name	SST	COL_2	NVI	DVI	ESO	SMI	COL_2	NVI	DVI	ESO	SMI
Java1.6	190	194	240	278	462	693	1.0	1.3	1.5	2.4	3.6
Geode	306	383	1099	911	1241	1905	1.3	3.6	3.0	4.1	6.2
Unidraw	15	15	15	15	33	42	1.0	1.0	1.0	2.2	2.9
Lov-obj-ed	37	49	82	97	123	184	1.3	2.2	2.6	3.3	4.9
SmartEiffel	54	54	90	92	121	295	1.0	1.7	1.7	2.2	5.5
PRMcl	38	40	49	62	102	152	1.1	1.3	1.7	2.7	4.0
Total	639	735	1574	1455	2081	3271	1.1	2.5	2.3	3.3	5.1
IBM-SF	394	553	805	668		2034	1.4	2.0	1.7		5.2
JDK1.3.1	142	147	174	191		536	1.0	1.2	1.3		3.8
Orbix	23	23	27	29		62	1.0	1.2	1.3		2.8
Corba	14	17	23	24		46	1.2	1.7	1.7		3.4
Orbacus	25	25	37	40		94	1.0	1.5	1.6		3.8
HotJava	25	26	30	32		99	1.1	1.2	1.3		3.9
Cecil	73	75	148	180		411	1.0	2.0	2.4		5.6
Dylan	71	71	84	118		311	1.0	1.2	1.6		4.4
Harlequin	23	23	49	59		146	1.0	2.1	2.6		6.3
Total	791	960	1379	1340		3739	1.2	1.7	1.7		4.7

The left part presents the total sizes for all considered techniques. The right part presents the ratios to the SST reference and the "Total" line represents the same ratios as for each benchmark, but computed from the sum of the corresponding parameters on all benchmarks, i.e. when a column depicts some p_b/q_b ratio for each benchmark b, the last line ratio is $\sum_b p_b / \sum_b q_b$.

The SMI case can be rewritten as $T_\theta = \sum_D n'_D M_D$, where n'_D is the number of D subclasses, including D. The ESO case amounts to excluding merged classes from the sum on D: $C \preceq_{\text{eso}} D$ means that $C \preceq D$ and there is no D' merged into D, i.e. empty and such that $C \preceq D' \prec_{\text{d}} D$. In the NVI case,

$$f_C^{nvi} = M'_C + \sum_{C \prec_{\text{d}} D} f_D^{nvi} \quad \text{with } M'_C = \begin{cases} M_C - M_D & \text{when } C \prec_{\text{e}} D \\ M_C = m_C & \text{otherwise } (C \text{ is a root}) \end{cases} \tag{12}$$

In the DVI case, the formula is more complicated:

$$f_C^{dvi} = f'_C + \sum_{C \prec_{\text{nnv}} D} f'_D \quad \text{where } f'_C = M'_C + \sum_{C \prec_{\text{nv}} D} f'_D \tag{13}$$

where $C \prec_{\text{nnv}} D$ iff $C \prec D$ and $\nexists D', C \preceq D' \prec_{\text{nv}} D$.

Table XII shows the total size, i.e. number of entries, of method tables in all techniques, together with the ratio to the ideal SST reference. The ESO to SST ratio is between 2 and 4. SMI confirms its high overhead, ESO and DVI their marked improvement and bidirectional coloring its very low overhead. When MI is intensively used, the difference between ESO and DVI is significant but less than expected: in fact, ESO improves upon SMI with the same ratio as DVI upon ESO. As for pure NVI, besides its unsound semantics, its bad worst-case complexity tends to reduce the gap with SMI and ESO.

Space Linearity. The criterion that is discussed in Section 2.4, page 13, relies on the assumption that method introduction, i.e. m_C, is uniform over all classes. Under this assumption, the average number $\overline{M_C}$ of methods known by a class would be $\overline{n_C} \times \overline{m_C}$, where $\overline{n_C}$ is the average number of superclasses. However, according

Table XIII. Estimate of number of method calls, and size of code, shifts and thunks

name	classes	methods	#*call*	code	table	shifts	thunks
Java1.6	5075	35352	141408	424	462	514	543
Geode	1318	14214	56856	171	1241	734	1872
Unidraw	614	3328	13312	40	33	43	36
Lov-obj-ed	436	5026	20104	60	123	102	171
SmartEiffel	397	7865	31460	94	121	123	134
PRMcl	479	3793	15172	46	102	82	130
Total	8319	69578	278312	835	2081	1597	2884

The first part of the table recalls the number of classes and method definitions ($m_C^{\mathtt{def}}$), then presents the number of calls. The second part presents, in the ESO case, the resulting code and table sizes, together with the extra space occupied specifically by shifts or thunks (in K-words).

Table XIV. Static space for coloring and subobject-based techniques, with thunks or shifts

	COL	thunks	shifts	Table
method table	T_θ	T_θ	T_θ	XII
upcast table	—	$U_\theta/2$	$U_\theta/2$	IX
receiver adjustment	—	$2(T_\theta - T_{\mathrm{sst}})$	$T_\theta/2$	
code size	3#*call*	3#*call*	5#*call*	XIII
total				XV

to Tables VII and X, $\overline{M_C}$ is markedly higher. This simply means that methods are preferably introduced in classes that are close to the hierarchy roots. It does not invalidate this criterion—for instance, the SMI overhead will be even higher since the root methods are markedly more repeated in subobject method tables.

Downcast. One should also add downcast tables ($\Delta^{\Uparrow}$). The statistics on perfect hashing (Section 4.2, page 27) presented in [Ducournau and Morandat 2009] show that these downcast tables would have about $4n_C$ entries for each class C, where n_C is the number of superclasses of C, including C (Table VII). In comparison, class coloring uses a little more than n_C entries. The ratio of about 4 is roughly the same as between SMI/ESO and COL for method tables. Therefore, we do not take downcast into account hereafter.

Code Size. The benchmarks do not provide any information about the number of method call sites, and for any other mechanism as well. Hence, there is no way to measure the static space occupied by the different techniques in the method code of each benchmark. However, several studies show that the number of call sites may be large enough to make code size significant w.r.t. table size. For instance, an average of about 4 call sites per method definition is reported in a SMALLTALK hierarchy [Driesen et al. 1995] and in the PRM compiler [Ducournau et al. 2009]. In the following, we assume a number of call sites $\#call = 4\sum_C m_C^{\mathtt{def}}$. Table XIII shows that the 3-instruction code sequence for method call in standard SST has a rather significant global impact on static space, even though Java1.6 statistics are overestimated because method declarations in interfaces are considered here as full definitions.

Receiver Adjustment. Shifts to receivers can be handled by thunks. The thunk number is exactly the difference $T_\theta - T_{\mathrm{sst}}$ in table size between the considered MI technique θ and SST. There is indeed a thunk per table entry, but for each class-method pair there is exactly one null shift. In this respect, the advantage

Table XV. Final comparison of static space between all considered techniques

	total size (in K-words)						ratio to SST				
name	SST	COL_2	DVI	shifts	thunks	SMI	COL_2	DVI	shifts	thunks	SMI
Java1.6	615	631	888	1419	1448	2155	1.0	1.4	2.3	2.4	3.5
Geode	476	563	2307	2191	3328	5323	1.2	4.8	4.6	7.0	11.2
Unidraw	55	56	56	117	110	140	1.0	1.0	2.1	2.0	2.6
Lov-obj-ed	98	112	277	290	359	543	1.1	2.8	3.0	3.7	5.6
SmartEiffel	148	150	264	341	352	879	1.0	1.8	2.3	2.4	5.9
PRMcl	83	86	157	231	279	428	1.0	1.9	2.8	3.4	5.1
Total	1474	1599	3949	4590	5877	9468	1.1	2.7	3.1	4.0	6.4
IBM-SF	1788	1987	2644			6893	1.1	1.5			3.9
JDK1.3.1	487	507	637			1699	1.0	1.3			3.5
Orbix	67	71	87			191	1.1	1.3			2.8
Corba	52	59	84			154	1.1	1.6			3.0
Orbacus	85	88	130			299	1.0	1.5			3.5
HotJava	66	69	88			293	1.0	1.3			4.4
Cecil	124	128	444			1147	1.0	3.6			9.3
Dylan	93	95	232			817	1.0	2.5			8.8
Harlequin	35	38	146			411	1.1	4.1			11.6
Total	2797	3043	4492			11904	1.1	1.6			4.3

The convention is the same as in Table XII, page App–21, but the numbers (in K-words) include method tables, code, shifts and thunks (only thunks for DVI and SMI, both for ESO).

of DVI over SMI lies in the null thunks which may be measured by the ratio $(T_{\text{dvi}} - T_{\text{sst}})/(T_{\text{smi}} - T_{\text{sst}})$. This ratio is around 40% in case of intensive use of MI (Geode and Lov), and the gain is in both static space and time. In contrast, without thunks, shifts occupy $T_\theta/2$ entries (with short integers) in method tables, plus 2 extra instructions in the code. Therefore, the static space occupied by explicit shifts in the code and the tables is $Z_{\text{shifts}} = 2\#call + T_\theta/2$. This must be compared with the cost of thunks, i.e. $Z_{\text{thunks}} = 2(T_\theta - T_{\text{sst}})$, since each thunk has two instructions.

Conclusion on Static Space. Finally, the static space consists of several pieces of memory: (i) method tables (T_θ); (ii) upcast tables (U_θ); (iii) receiver adjustment, with thunks or shifts; and (iv) code length for all call sites. The total static size is then obtained with $3\#call + T_\theta + Z + U_\theta/2$. Table XIV presents these statistics with thunks for all techniques, but for ESO for which both shifts and thunks are displayed. These statistics show that, even when taking the code size into account, which is advantageous for thunks, the thunk space overhead remains greater.

Overall, the overhead of subobjects is even higher for static than for dynamic space. On average on the first group of benchmarks, the static space occupied by the data structures and code sequences that implement method invocation and subtype testing are 6-fold larger with standard subobject-based implementation than with coloring. Once again, empty-subobject optimization is a middle point with a ratio about 3 or 4. In contrast, subobjects hardly double dynamic space (Table VII, page App–17).

In these estimations, $T_\theta/2$ and $U_\theta/2$ rely on the assumption that the processor has a specific half-word `load` instruction. This is not a common feature, however, and many processors will require an extra shift or bit-wise mask. This small increase in code length would partially offset the half-word benefits. Hence full word data are likely preferable, thus reducing the difference between shifts and thunks.

Table XVI. Driesen's pseudo-code

`R1`	a register (any argument without `#`)
`#immediate`	an immediate value (prefix `#`)
`load [R1+#imm], R2`	load the word in memory location `R1+#imm` to register `R2`
`store R1,[R2+#imm]`	store the word in register `R1` into memory location `R2+#imm`
`add R1, R2, R3`	add register `R1` to `R2`. Result is put in `R3`
`and R1, R2, R3`	bit-wise `and` on register `R1` and `R2`. Result is put in `R3`
`call R1`	jump to address in `R1` (can also be immediate), save return address
`comp R1, R2`	compare value in register `R1` with `R2` (`R2` can be immediate)
`bne #imm`	if last compare is not equal, jump to `#imm` (`beq`, `blt`, `bgt` are analogous)
`jump R1`	jump to address in `R1` (can also be immediate)

D. PSEUDO-CODE SURVEY

We present an evaluation of time efficiency based on a model of modern processors and on a pseudo-code, both borrowed from [Driesen and Hölzle 1995; 1996; Driesen et al. 1995; Driesen 2001]. Table XVI recalls the intuitive instruction set of this assembly language [Driesen 2001, p. 193]. Modern processors are mainly characterized by a pipe-line architecture with instruction-level parallelism and branch prediction. Each machine instruction takes a certain number of clock cycles. Whereas bit-wise and simple arithmetic instructions are 1-cycles, most other instructions involve a latency of several cycles. Therefore, each code sequence will be measured by an estimate of the number of cycles, parametrized by memory `load` latency—L whose value is 2 or 3—and branching latency—B whose value may run from 3 to 15. In contrast, a `store` takes 1 cycle. Note that both latencies assume that the data is cached, i.e. a cache miss would add up to 100 cycles.

The processor specifications that we retain are mostly the same as processor P95 in Driesen [2001]. Particularly, 2 `load` instructions cannot be executed in parallel and need a one cycle delay. Moreover, the maximum number of integer instructions per cycle is 2. The cycle measure of each example is given as a function of L and B. This model should roughly apply to all modern processors for general purpose computers (PC, laptops, workstations and servers). In contrast, processors for embedded systems are markedly less powerful and their architecture does not always provide pipe-line, instruction-level parallelism and prediction. We just present and discuss in this appendix some simple examples from Sections 2, page 9, and 3, page 17. More details and examples can be found in the work of K. Driesen and in [Ducournau 2008; Ducournau et al. 2009].

D.1 Single Subtyping

Method call:

```
load [object + #tableOffset], table
load [table + #methodOffset], method                    2L + B
call method
```

`#tableOffset` is a constant for all types and methods whereas `#methodOffset` depends on the method (Invariant 2.2). Moreover, `#tableOffset` will likely be 0 in most implementations.

Attribute access:

```
load [object + #attributeOffset], attribute                    L
```

`#attributeOffset` depends on the attribute (Invariant 2.2).

Cohen's display yields the following sequence for subtype testing:

```
load [object + #tableOffset], table
load [table + #targetColor], classId
comp classId, #targetId                                        2L + 2
bne #fail
// succeed
```

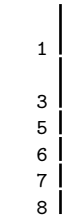
`#targetColor` and `#targetId` depend on the target class (Invariant 2.2). `#fail` is the address of the code which either signals an exception or executes what must be done; it may be shared by several, or even all, type checks, or specific to each one.

Schubert's numbering is a little more complicated. Under the CWA, the following sequence works:

```
load [object + #tableOffset], table
load [table + #n1Offset], classId
comp classId, #n1
blt #fail                                                      2L + 4
comp classId, #n2
bgt #fail
// succeed
```

`#n1Offset` is a constant, whereas `#n1` and `#n2` depend on the target class. However, the two conditional branchings increase misprediction risks, but an arithmetic trick, presented in the next sequence, avoids it. Under the OWA, `n1` and `n2` require an indirection, and the sequence is markedly longer:

```
1   load [object + #tableOffset], table
2   load [#n1Address], n1
3   load [table + #n1Offset], classId
4   load [#n2Address], n2
5   sub classId, n1, n1
6   sub classId, n2, n2
7   xor n1, n2, r
8   bgz #fail
    // succeed
```



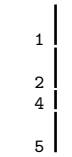
$2L + 4$

This code sequence is complicated enough to involve some instruction-level parallelism. When this is the case, the cycle measure is not straightforward and a diagram proposes a possible schedule. There are here two independent `load` sequences which can run in parallel, and the total cycle count is not greater than in the previous sequence. However, these extra `loads` increase the cache-miss risk.

D.2 Multiple Inheritance

All instructions that are not used in SST are italicized. A method call requires a pointer adjustement on the receiver:

```
1   load [object + #tableOffset], table
2   load [table + #methodOffset+1], delta
3   load [table + #methodOffset], method
4   add object, delta, object
5   call method
```



$2L + B + 1$

`#methodOffset` depends on both the method and the static type of the receiver.

With a thunk, the call sequence is the same as with SST, and the thunk consists of an addition before a static branching:

```
add object, #delta, object
jump #method
```

`#delta` and `#method` depend on both the method and the static type of the receiver and the call sequence is that of single subtyping.

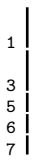
Upcast from type T to U, with $T <: U$:

```
load [object + #tableOffset], table
load [table + #castOffset], delta                    2L + 1
add object, delta, target
```

`#castOffset` is the position of $\Delta_{T,U}$ that depends on both T and U. Attribute access in the general case involves also an upcast to the class introducing the attribute (T_p) for a total of $3L + 1$ cycles.

An equality test with unrelated types involves two pointer adjustments, with some parallelism:

```
1   load [x + #tableOffset], table1
2   load [y + #tableOffset], table2
3   load [table1 + #dtypeOffset], dx
4   load [table2 + #dtypeOffset], dy
5   add dx, x, x
6   add dy, y, y
7   comp x, y
```

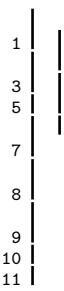


$2L + 3$

`#dtypeOffset` denotes the constant position of $\Delta^{\Downarrow}$. When both types are related by subtyping, this simplifies to an upcast followed by a comparison ($2L + 2$ cycles).

Finally, when all static types differ, the assignment $a.x := b.y$ is quite intricate:

```
 1   load [objecta + #tableOffset], tablea
 2   load [objectb + #tableOffset], tableb
 3   load [tablea + #castaOffset], deltaa
 4   load [tableb + #castbOffset], deltab
 5   add objecta, deltaa, objecta
 6   add objectb, deltab, objectb
 7   load [objecta + #attributeaOffset], attribute
 8   load [attribute + #tableOffset], table
 9   load [table + #castOffset], delta
10   add attribute, delta, attribute
11   store attribute, [objectb + #attributebOffset]
```



$5L + 3$

By comparison, two instructions of `load` and `store` are sufficient with SST implementation.