Perfect Hashing as an Almost Perfect Subtype Test

ROLAND DUCOURNAU
LIRMM – CNRS and Université Montpellier II, France

Subtype tests are an important issue in the implementation of object-oriented programming languages. Many techniques have been proposed, but none of them perfectly fullfills the five requirements that we have identified: constant-time, linear-space, multiple inheritance, dynamic loading and inlining. In this paper, we propose to apply a well known technique, *perfect hashing*, and we show that it responds rather well in meeting these requirements. Furthermore, in the framework of JAVA like languages—characterized by single inheritance of classes and multiple subtyping of interfaces—perfect hashing also applies, at the same time, to method invocation when the receiver is typed by an interface.

The linear-space criterion is validated by statistical simulation on benchmarks constituted of large-scale class hierarchies.

Categories and Subject Descriptors: D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; JAVA; EIFFEL; D.3.3 [**Programming languages**]: Language Constructs and Features—*datatypes and structures*; *procedures, functions, and subroutines*; D.3.4 [**Programming languages**]: Processors—*compilers*; *linkers*; *loaders*; *run-time environments*; E.1 [**Data**]: Data Structures—*lists*; *tables*; E.2 [**Data**]: Data Storage Representations—*Hash-table representations*

General Terms: Languages, Measurement, Performance

Additional Key Words and Phrases: casting, coloring, downcast, dynamic loading, interfaces, linking, method tables, multiple inheritance, multiple subtyping, perfect hashing, single inheritance, subtype test, virtual function tables

1. INTRODUCTION

The need for dynamic subtyping checks is an original feature of object-oriented programming. Given an entity x of a static type C, the programmer or compiler may want to check that the value bound to x is an instance of another type D, generally before applying to x an operation which is not known by C. C and D are, respectively, the *source* and *target* types and D is usually a subtype of C. In the most common object-oriented framework, C and D are classes and D is a subclass of C (denoted $D \preceq C$). In a dynamically typed language, x is statically untyped—hence, the source type is the root $\top$ of the type hierarchy. Moreover, with multiple inheritance, D is not always a subtype of C. However, these two exceptions have no effect on the considered mechanism, which only involves the value bound to x and the target type D.

Subtyping checks may be *explicit* or *implicit*. Programming languages offer various syntactic constructs that enable the programmer to make *explicit* checks. *Downcast* is commonly used by C++, JAVA and C# programmers, according to various syntactic forms: `dynamic_cast` in C++ [Stroustrup 1998], parenthesized syntax in JAVA (a C syntax which must not be used in C++!), `typecase` in THETA [Liskov et al. 1995] or *assignment attempts* in EIFFEL [Meyer 1992; 1997]. The com-

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

mon point is that the target type is a constant of the construct. Downcast failures may be treated in several ways, either by signaling an exception (JAVA, C++ for references only), returning a null value (EIFFEL, C++ for pointers only), or in a boolean way (`typecase`).

When the language is equipped with some *Meta-Object Protocol* (MOP)—like CLOS [Steele 1990; Kiczales et al. 1991], SMALLTALK [Goldberg and Robson 1983] or JAVA—the target type may be computed from some other values. For instance, the programmer may want to check that the value bound to x is actually an instance of the class of the value bound to another entity y. Such type checks are handled by boolean operators like `isInstanceOf`, e.g. `x.isInstanceOf(y.class)`.

Finally, subtype checks may be *implicit*, when they are generated by the compiler without being intended by the programmer. This is the case as soon as some language feature is not *type safe*. For instance, array subtyping in JAVA and covariant overriding in EIFFEL are well known unsafe features, which may be valuable [Meyer 1997; Ducournau 2002b][1] but require dynamic type checking.

Once again, the difference between these cases—according to whether the target type is *static* or *dynamic*[2], or the check is *implicit* or *explicit*—is not essential. The only point is that, when the target type is static, the data concerning it may be treated as an immediate value, thus avoiding memory access.

Without loss of generality, subtyping test implementation amounts to some encoding of the class hierarchy. Despite many works from the beginning of object-oriented programming [Schubert et al. 1983; Agrawal et al. 1989; Aït-Kaci et al. 1989; Cohen 1991; Caseau 1993; Habib and Nourine 1994; Habib et al. 1995; Habib et al. 1997; Krall et al. 1997; Vitek et al. 1997; Alpern et al. 2001; Raynaud and Thierry 2001; Zibin and Gil 2001; Click and Rose 2002], general and efficient implementation of subtype checks remains an open issue. To our knowledge, no implementation meets the following five requirements: i) truly *constant-time*, ii) as much *linear-space* as possible, iii) compatible with *multiple inheritance*, iv) and with separate compilation and *dynamic loading* and, finally, v) with a code sequence short enough to be *inlined*. i) Constant-time implementation of frequently used mechanisms is a major requirement. Of course, non-constant time implementations may be better when they are better on average, but this must be carefully proven. ii) Object-oriented mechanisms are commonly implemented using tables associated with classes—the total size of these tables should be linear in the size of the inheritance graph. Linearity in the number of classes is actually not possible—usual implementations are, in the worst case, quadratic in the number of classes, but linear in the size of the inheritance relationship (transitive closure). Moreover, this requirement means that the size occupied by a class is linear in the number of its superclasses. In contrast, C++ implementation is cubic in the number of classes [Ducournau 2005]. iii) Complex real-world models—one should now say *ontologies*—require multiple inheritance, at least in the weak form of single inheritance of classes but *multiple subtyping* of interfaces, as in JAVA and DOTNET

[1] The existence of downcast features in type safe languages is a counter-argument to the claim that type safety is ensured by these languages and an argument for unsafe features like covariance.
[2] These terms are unrelated with the C++ keywords `static_cast` and `dynamic_cast`, which are both *static*.

Table I. Subtyping techniques compared to functional and efficiency requirements

	i	ii	iii	iv	v
[Schubert et al. 1983]	x	x	–	–	x
[Cohen 1991]	x	x	–	x	x
[Vitek et al. 1997]	x	x	x	–	x
[Zibin and Gil 2001]	x	x	x	–	x
[Alpern et al. 2001]	x	–	x	x	x
[Click and Rose 2002]	–	x	x	x	x
C++ (g++)	–	?	x	x	–
SMART EIFFEL	–	x	x	–	?
direct access	x	–	x	x	x
perfect hashing	x	x	x	x	x

languages like C# [Microsoft 2001]. The fact that there is almost no statically typed language in pure single inheritance, i.e. in *single subtyping*, is a strong argument. iv) Dynamic loading is an old feature of LISP and SMALLTALK platforms, before becoming a key requirement of modern runtime systems for the Web, in JAVA and DOTNET languages. Note that a more traditional language like C++ is quite compatible with dynamic (or incremental) linking. Therefore, dynamic loading require incremental algorithms for encoding the class hierarchy. v) Finally, inlining is a classic optimization which avoids function calls and improves time efficiency at the expense of a sligth increase of code size. It is a key optimization for frequently used basic mechanisms, provided that the inlined code sequence is small and simple enough. In the context of table-based object-oriented implementations, inlining has the advantage that one access to the method table may serve for several successive operations, e.g. subtype test followed by method invocation. In contrast, the GNU C++ compiler `g++` compiles `dynamic_cast` into a function call. Inlining is closely related to the incrementality of the encoding algorithms. Indeed, one might argue—and some authors do—that a fast global (i.e. non-incremental) algorithm may be compatible with dynamic loading, since fastness allows complete recomputation at each class loading. However, in this case, *static* subtyping checks could not use immediate values but should make some memory access to the data structure of the target type, which should decrease time-efficiency and increase the number of inlined instructions. Table I compares some known techniques, discussed hereafter, with the five criteria.

In this paper, we present a subtyping test implementation which has the advantage of fullfilling our five requirements. Our proposition is based on a 30-year old technique known as *perfect hashing* [Sprugnoli 1977; Mehlhorn and Tsakalidis 1990], which is an optimized and constant-time variant of more traditional hashtables like *linear probing* [Knuth 1973; Vitter and Flajolet 1990].

The structure of the paper is as follows. Section 2 first presents the well known Cohen [1991] test, which fullfills all requirements but multiple inheritance, and then an extension to multiple inheritance [Vitek et al. 1997] which no longer allows dynamic loading. Different notions are introduced on the way. Section 3 describes simple hashtable-based implementations and shows how they are improved with *perfect hashing*. The application to C++ subobject-based implementation is detailed, along with JAVA `invokeinterface`. Section 4 proposes some statistics on commonly used object-oriented benchmarks and presents the choice between two hash functions as a tradeoff between time and space efficiency. Applications to

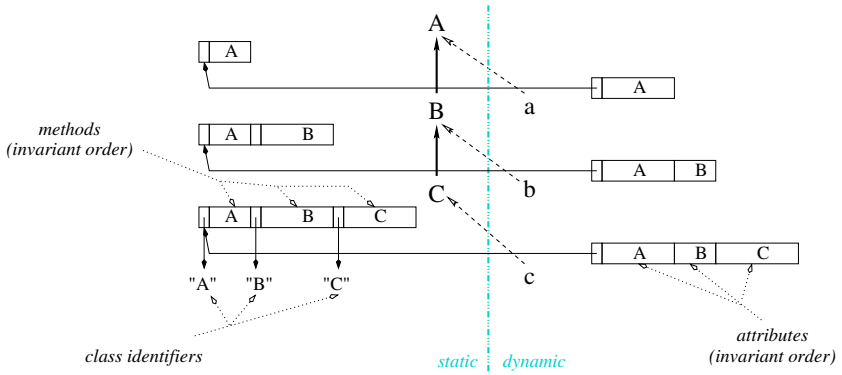


Fig. 1. Object layout (dynamic memory, right) and method tables (static memory, left) in single subtyping: 3 classes A, B and C with their respective instances, a, b and c.

C++ and JAVA are examined. Section 5 presents some other techniques and shows how they fullfill—or not—our five requirements. In the conclusion, we examine how to improve this use of perfect hashing together with some other applications.

2. MULTIPLE INHERITANCE VS. DYNAMIC LOADING

One of the simplest subtyping check works well in single inheritance but its natural extension to multiple inheritance is no longer compatible with dynamic loading.

2.1 Single inheritance

2.1.1 *Basic object implementation.* In separate compilation of statically typed languages, late binding is generally implemented with method tables (aka *virtual function tables* in C++ jargon). Method calls are then reduced to function calls through a small fixed number (usually 2) of extra indirections. On the other hand, an object is laid out as an attribute table, with a header pointing at the class table and some added information, e.g. for garbage collection. With single inheritance, the class hierarchy is a tree and the tables implementing a class are a straightforward extension of that of its single superclass (Figure 1). The resulting implementation respects two essential *invariants*: i) a *reference* to an object does not depend on the static type of the reference; ii) the *position* of attributes and methods in the table does not depend on the dynamic type of the object. Therefore, all accesses to objects are straightforward. Note that this simplicity is due to both static typing and single inheritance—dynamic typing adds the same kind of complication as multiple inheritance, since the same property name may be at different places in unrelated classes.

2.1.2 *Subtype test.* Note first that, contrary to object layout and method tables, the subtype test does not depend on static or dynamic typing. In the single inheritance framework, many techniques have been proposed. We only present two of them as they are the simplest ones. The first one meets all five requirements but multiple inheritance, and it is the basis for further generalizations. The second one is constant-space but it has some drawbacks.

Notations and definitions. $\prec$ denotes the class specialization relationship, which is transitive, antireflexive and antisymmetric, and $\preceq$ is its reflexive closure. Note that we identify here types to classes and subtyping to specialization. Given an object, τ_d denotes *the* class of the object (aka its *dynamic type*, i.e. the class which produced the considered object by instanciation. The key characteristics of specialization is that $C \prec D$ implies that all instances of C are instances of D. Therefore, the object is instance of another class C iff $\tau_d \preceq C$. In a static typing framework, the static type of the reference to the object, denoted τ_s, is a supertype of the instanciation class: $\tau_d \preceq \tau_s$.

2.1.2.1 *Cohen's technique.* The technique was first described by Cohen [1991] as an adaptation of the "display" originaly proposed by [Dijkstra 1960]. It has been widely reused and rediscovered by different authors (e.g. [Queinnec 1997; Alpern et al. 2001]). 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. Given an object of dynamic type τ_d, this object is an instance of a class C iff the object's method table, denoted tab_{τ_d}, contains, at offset $\chi(C)$ (called the *color* of C), the identifier id_C:

$$\tau_d \preceq C \Leftrightarrow tab_{\tau_d}[\chi(C)] = id_C \tag{1}$$

Class offsets are ruled by the same position invariant as methods and attributes. In fact, one may understand this as if each class introduces a method for subtyping tests, i.e. such that its instances can check they actually are.

The code for testing this condition is quite straightforward:

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

`Object` is the register which holds the object address, `tableOffset` is the offset of the pointer to method table in the object layout (usually 0), `targetColor` is the offset of the target class, and `targetId` the target class identifier. '`#`' denotes immediate values. This intuitive pseudo-code is borrowed from [Driesen 2001]—where the reader may find, and also in [Ducournau 2005], more information on its interpretation. In terms of processor cycles, the execution of the sequence would take $2L + 2$ cycles, where L is the *load latency*, which is around 2-3 and does not consider *cache misses* (up to 100 cycles). Modelling cache misses would be hard work, out of the scope of this paper. However, it should be kept in mind that not all `load`s risk cache misses, since several successive `load`s from the same memory area should entail only one cache miss. Note that this is not the case for the previous code, as the two `load`s are clearly from disconnected areas.

A point must be carefully examined. In theory, $tab_{\tau_d}[\chi(C)]$ is sound only if $\chi(C)$ does not run out of the bounds of tab_{τ_d}. If one assumes that class offsets are positive integers, a comparison of $\chi(C)$ with the length of tab_{τ_d} seems to be required, together with memory access to the length itself—this would hinder efficiency. Fortunately, there is a simple way to avoid this test, by ensuring that, in some specific memory area, the value id_C always occurs at offset $\chi(C)$ of some *tab*. Consider

that method tables contain only method addresses and class identifiers and that they have contiguous allocations in this specific memory area, which therefore contains only addresses and identifiers. As addresses are even numbers (due to word alignment), coding class identifiers with odd numbers would avoid any confusion between the two types. Finally, the specific memory area must be padded with some even number, to a length corresponding to the maximum *tab* size[3]. Nevertheless, method tables might contain more data than addresses and identifiers, i.e. something that might take any half-word or word value—even though we did not identify what. Therefore, either a more complex coding or an indirection might be required. So, if class identifiers are gathered within specific tables, distinct from method tables and allocated in the same contiguous way, this extra indirection will have the same cost as access to length—apart from cache misses—but the test itself will be saved. A more common and frequently proposed way to save on this extra test is to use fixed size tab_{τ_d}, at the expense of a large space overhead[4].

Clearly, the technique fullfills all requirements but multiple inheritance: i) constant-time, ii) linear-time, iv) dynamic loading and v) inlining. Dynamic loading requires some explanations. First, a class identifier must be assigned to each class when it is loaded. Simple numbering is the solution, sometimes using only odd or even numbers, or negative ones, according to the coding which has been decided. Classes are assumed to be loaded in a natural specialization ordering, i.e. C before D when $D \prec C$. Secondly, all offsets and identifiers (`targetColor` and `targetId`) must appear as symbols in the code generated by the compiler, and must be replaced by values at link-time. Of course, the approach is also feasible in a global linking framework, provided that the linker is equipped with such specific but simple functionalities.

Regarding space, the occupation is clearly linear as it is a one-to-one coding of the inheritance partial order—i.e. each inheritance pair uses one entry in the tables. A simple improvement consists of assigning two classes to the same word, since short integers are sufficient for coding class identifiers. However, the overall cost of Cohen's test is not negligible. As already stated, this subtype test implementation conceptually adds one method per class, but it needs only a half word per entry. Now statistics on method numbers (see Table III) show that, in average, the number of methods introduced by a class runs between 0.5 and 15. In the latter case, the relative overhead would be meaningless but not in the former, though total sizes are not directly related with the number of introduced methods. More space-efficient variants exist, based on a compact, variable length, coding of class identifiers, which is no longer absolute but relative to classes with the same color (i.e. depth). However, this is to the detriment of: i) separate compilation, or

[3]In the dynamic loading framework, this maximum size is not known—an upper bound must be used, which is a parameter of the runtime platform. Moreover, there could be more than one method table area—when it is full, a new one may be allocated, and the maximum size may also be adjusted, without exceeding the padded size in the previous areas.

[4][Click and Rose 2002] attributes the technique to [Pfister and Templ 1991]. In Section 4, Table II shows that the maximum superclass number may be 5-fold greater than its average. Note that an arbitrary upper bound does not entail the same overhead, according to whether it is used for each class, as here, or only for one or a few specific areas, as in the previous note.

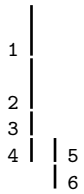
at least dynamic linking, ii) time efficiency and code size, as colors are no longer aligned to words or half words.

2.1.2.2 *Schubert's numbering.* This 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:

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

This technique, due to Schubert et al. [1983], is also called *relative numbering*. Only two short integers are needed 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 test is static—they can be compiled as constants. The sequence code for a static check would be:

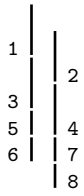
```
load [object + #tableOffset], table
load [table + #n1Offset], classid
comp classid, #n1
blt #fail
comp classid, #n2
bgt #fail
// succeed
```



$2L + 3$

Note that some parallelism is possible, which saves on one cycle, and a possible schedule is given by the diagram on the right. `N1Offset` is a constant. In a global setting, `n1` and `n2` can be immediate values. However, when used in a dynamic loading framework, this technique requires a complete computation each time a class is loaded—this is possible since the complexity is linear in the number of classes—and `n1` and `n2` require memory accesses. Hence, the code would be changed into the following:

```
load [object + #tableOffset], table
load n1Address, n1
load [table + #n1Offset], classid
load n2Address, n2
comp classid, n1
blt #fail
comp classid, n2
bgt #fail
// succeed
```



$2L + 3$

This adds two `load` instructions but no cycles as they are partly run in parallel (see also note 7, page 11). However, executing instructions in parallel does not nullify their overhead as they possibly occupy the place of some other instructions. Moreover, the two extra `load`s add one extra cache miss risk.

2.2 Multiple inheritance

2.2.1 *Standard implementation.* With multiple inheritance, the two invariants of reference and position, which hold with single inheritance and static typing, cannot hold together. Therefore, the "standard" implementation of multiple inheritance—i.e. that of C++—is based on *subobjects*. The object layout is composed of several subobjects, one for each superclass of the object's class (τ_d). Each subobject con-

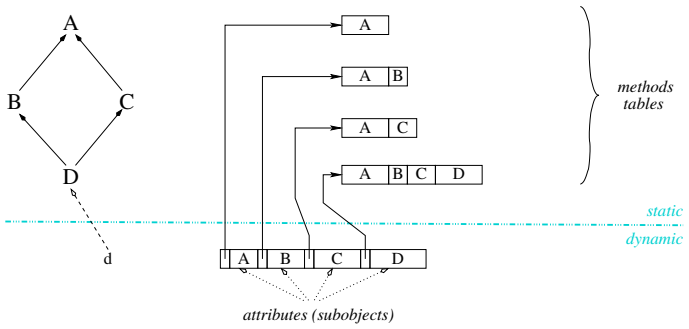


Fig. 2. Object layout and method tables with multiple inheritance: an instance d of the class D is depicted

tains attributes introduced by the corresponding class, together with a pointer to a method table which contains the methods known—i.e. defined or inherited—by the class (Figure 2). A reference of a given static type (τ_s) points at the subobject corresponding to this type. This is the C++ implementation, when the keyword `virtual` annotates each superclass [Ellis and Stroustrup 1990; Lippman 1996; Ducournau 2005]. It is constant-time and compatible with dynamic loading, but method tables are no longer linear-space, but quadratic. Furthermore, all polymorphic object manipulations, which are quite numerous, need pointer adjustments between source type and target type, as they correspond to different subobjects[5].

As a consequence on our focus of interest, the subtyping test is no longer a boolean operation, i.e. it must also return, in case of success, the shift which must be added to the address of the source type subobject to get the target address. Therefore, if one considers Cohen's type implementations, the table entries must be twofold: a class identifier and a shift. Of course, one can easily imagine naive techniques for implementing a subtype test, e.g. hashtables. They are presumed to not meet our first requirement, i.e. constant-time. We will examine them in Section 3 but another approach must be examined before ruling it out.

Direct access. A technically simple solution allowing direct access is a $N \times N$ matrix, where N is the class number: $mat[id_T, id_U]$ contains $\Delta_{T,U}$, the shift between subobjects T and U, if $U \prec T$, and otherwise a distinguished value meaning 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 $N >> 1000$. Class identifiers id_C can be computed at load time, in the load ordering, i.e. $id_C \geq id_D$ when $C \preceq D$ and the numbering is a *linear extension* of $\preceq$. The square matrix may then be replaced by a triangular one, i.e. by one vector per class, using $vect_U[id_T]$ instead of $mat[id_T, id_U]$. The average cost is reduced to N bytes per class—this is the worst case cost of Cohen's technique, when the class hierarchy is a linear chain, but its practical cost is far lower. Clearly, this approach must be ruled out since it would not be linear-space, at least in the framework of plain multiple inheritance.

[5]Note that these pointer adjustments are safe—i.e. the target type is always a supertype of the source type—and are implemented more simply than subtyping tests.

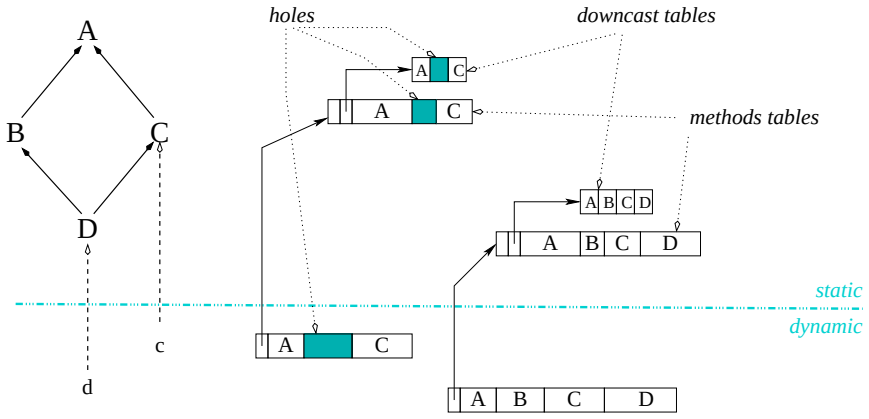


Fig. 3. Unidirectional coloring heuristics applied to classes, methods and attributes. Class coloring appears in separate tables for a better view—they are actually merged within method tables.

With JAVA-like multiple subtyping, the conclusion would be somewhat different (see Section 3.4).

2.2.2 *Class, attribute and method coloring.* We now detail the coloring approach as it is quite versatile and naturally extends the single inheritance implementation to multiple inheritance, while meeting all five requirements except compatibility with dynamic loading.

Method coloring was first proposed by Dixon et al. [1989] under the name of *selector coloring*. [Pugh and Weddell 1990; Ducournau 1991] applied *coloring* to attributes and Vitek et al. [1997] to classes (under the name of *pack encoding*).

The general idea of coloring is to keep the two invariants of single inheritance—i.e. reference and position. However, this cannot be done separately for each class, but globally for the complete hierarchy. An injective numbering of attributes, methods and classes verifies the invariant, so this is clearly a matter of optimization for minimizing the size of all tables. The minimization problem is akin to graph coloring [Garey and Johnson 1979] and proven to be NP-hard [Pugh and Weddell 1993; Takhedmit 2003]. Therefore heuristics are needed and some experiments by Pugh and Weddell [1990] Ducournau [1997; 2002a] and Takhedmit [2003] have shown their efficiency and that the technique is tractable. [Ducournau 2006] surveys the approach.

Class coloring—aka *pack encoding* [Vitek et al. 1997]—is thus the natural extension of Cohen's [1991] technique. Regarding the requirements, v) code sequence and i) time efficiency are exactly the same; ii) the table sizes are slightly increased (see Section 4), but the overhead is much lower than with subobject-based implementation; but iii) dynamic loading is not supported. Actually, we have exchanged multiple inheritance for dynamic loading.

3. HASHTABLE-BASED APPROACH

A simple naive way to implement the subtyping test is one *hashtable* per class, associating with the identifier of each superclass a shift to the corresponding subobject. In a reference-invariant implementation, the hashtable may be simplified into a

hashset, since all shifts are null. We present hereafter how two classic hashtables might be used for a subtyping test, then perfect hashing.

3.1 Linear probing

Among the various hashtable implementations, a simple one is *linear probing* [Knuth 1973; Vitter and Flajolet 1990]. With linear probing, a hashtable is a an array of H entry pairs, alternatively keys in interval $[0..N-1]$ and values, together with a function $h : [0..N-1] \to [0..H-1]$. A key x is sequentially searched in the array, starting from position $h(x)$, until either x is found (success) or a distinguished key $\perp$ (e.g. a negative integer) is found (failure). When the end of the array is found, the search continues at index 0. Experiments and statistical analyses show that the average number of probes is close to 1 when n/H is small enough, where n is the number of occupied entries. A small variant avoids testing the array end by unfolding this circular search, so the array may have more than H entry pairs, up to $H + n$. This improves the code and its efficiency, with minimal space overhead.

In the subtyping test context, such a hashtable is associated with each class C, and the number n of keys is the number n_C of C superclasses (including C). Moreover, with this variant, H can be lesser than n, but this would be to the detriment of time efficiency. Therefore, H cannot be the same for all classes—i.e. the space would be wasted for classes with a small n_C/H, and time efficiency would be low when n_C/H is high. Obviously, the constant-time requirement would not be ensured. Hence, a parameter H_C must be set for each class, for instance $H_C = 2n_C$, and stored in the method table. Consequently, these hashtables are of varying length and they can be inlined in the method table at negative offsets only.

A last point must be decided, the hashing function h. Of course, it depends now on class C, so we need a family of functions h_C, parameterized by H_C, hence $h_C(x) = hash(x, H_C)$. The function *hash* must be simple enough to be inlined—a one-instruction function would be better and a one-cycle instruction the best. Moreover, *hash* is a function from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$ such that, $\forall x \in \mathbf{N}, hash(x, y) \leq y$. Two simple functions are good candidates, i.e. bit-wise `and` and *modulus* (remainder of the integer division, denoted `mod`). The former seems better as it is a one-cycle instruction, whereas integer division takes several cycles[6].

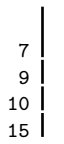
Finally, with bit-wise `and`, the code for the subtyping test is the following 15-instruction sequence, where `HtOffset` is the offset of the hashtable in the method table and `hashingoffset` is the offset of H_C—both do not depend on C:

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
add table, #htOffset, table
and #targetId, h, h
mul h, 4, h
add table, h, table
```

1	3	$2L+3$
2		
4		
5		
6		

[6]The latency of integer division varies markedly according to processors and data. For instance, in Intel Pentium IV, integer division uses the micro-coded floating-point unit, and may run from some cycles to several tens. In the following, we assume an hypothetically low latency of 6 cycles.

```
loop:    load table, id
         load [table + #2], delta
         comp #targetId, id
         beq #succeed
         comp #empty, id
         beq #fail
         add table, #4, table
         jump #loop
succeed: add object, delta, object
```



$L+3$

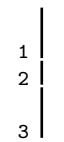
The class identifier is hashed then searched in the hashtable, starting from the hashed value, until a value `empty` is found. The two italicized instructions can be saved. In line 3, `htOffset` could be 0—only on condition that the hashtable is inlined at negative offsets in the method table and `sub` replaces `and` at line 6. In line 5, which serves for word alignment, both `targetId` and bit-wise mask `h` could be multiples of 4. In the following, we consider that these instructions are saved.

There are 13 instructions. This is not very short for inlining and more than 3-fold longer than class coloring. In terms of cycles, the diagram shows a possible schedule on a modern processor—note how the last `load` (instruction 8) is done in parallel, saving on L cycles[7]. The resulting cycle number is $2L+k(L+5)$ in case of failure—add one cycle in case of success—where k is the number of probes, greater than 1 and less than n_C (success) or n_C+1 (failure). On average, k should be close to 1, so it would be almost 2 times slower than class coloring (with $L = 2$ or 3). With modulus, the code would be exactly the same, except that integer division would be substituted for bit-wise `and`, but about 5 cycles would be added.

3.2 Separate chaining

An alternative to linear probing is *separate chaining* [Knuth 1973; Vitter and Flajolet 1990]. The hashtable is a simple fixed-size array, whose entries point at chained lists of key-value pairs. In our case, the list may be an array—this avoids explicit chaining. With separate chaining, the table size may be less than the number of items ($H < n$) and any table size can work with an average efficiency. Therefore, a uniform hash parameter H is possible, which allows compile-time hashing and simplifies the code:

```
         load [object + #tableOffset], table
         add table, #hid, table
         load table
loop:    // same as for linear probe
```



$2L+1$

`Hid` is the sum of `htOffset` and of the hash value of `targetId`. However, the gain is not significant—1 cycle and 1 instruction—since the load of the hash parameter is replaced by the load of the chaining array, whose locality is not better. Moreover, the constant-time requirement is not ensured, even though the efficiency is better

[7] An equivalent gain would be obtained with processors like Pentium, which allow accesses to subregisters—the two 2-byte `loads` would be replaced by one 4-byte `load` and subregisters would be used in subsequent instructions. This would also save on one instruction.

than with static linear probes, since linear searches are uniformly distributed on all hashtable entries, whereas with linear probes, the whole table may be scanned.

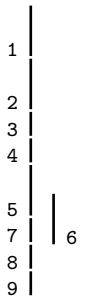
An important optimization involves compiling each chaining into a small binary tree dispatching code. However, it works well for method invocation [Alpern et al. 2001], but here it would require an explicit function call, to the detriment of inlining. It also presumes that code generation is possible at load-time.

3.3 Perfect hashing

Note that the considered hashtables have a key particularity, i.e. they are *constant hashtables*—once they have been computed, at link or load time, there is no longer any need for insertion or deletion[8]. Therefore, for each class C, knowing the identifiers of all its superclasses, it is possible to optimize the hashtable in order to minimize the table size and the average number of probes, either in positive or negative cases. In the ideal case from a time standpoint, H_C may be defined in such a way that all tests only need one probe, i.e. h_C is injective on the set of identifiers of all superclasses of C. This is known as a *perfect hashing* function [Sprugnoli 1977; Mehlhorn and Tsakalidis 1990]. Note that we are not searching a *minimal perfect hashing function*—i.e. a function such that $H_C = n_C$—because such a function would be more complicated to both compute and inline.

On the whole, for each class C, we define and compute H_C as the minimal positive integer such that h_C is a perfect hashing function on the set of identifiers of the superclasses of C. Clearly, $H_C \geq n_C$ for both hash functions and $H_C \leq id_C$ (resp. $H_C < 2id_C$) with `mod` (resp. `and`). We did not conduct an in-depth mathematical analysis, but instead let simulations prove the feasibility of the approach (see Section 4). The table size is exactly H_C (resp. $H_C + 1$)[9] with modulus (resp. bit-wise `and`) and the code may be simplified into the following 9-instruction sequence:

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #targetId, h, h
sub table, h, table
load table, id
load [table + #2], delta
comp #targetId, id
bne #fail
add object, delta, object
```



$3L + 5$

The cycle number is almost the same as with linear probing: $3L + 4$ in the failure case, plus one cycle when the test succeeds. This remains two fold the instruction number and almost two fold the coloring cycle number, but it is likely very close to optimum for standard MI implementations.

A last point must be discussed. The previous code presumes that each subobject has a hashtable in its method table. This would be space expensive, i.e. cubic. So

[8]Unloading or reloading classes is not part of our requirements. Anyway, it should be possible to specify it in a way that is compatible with the present approach.

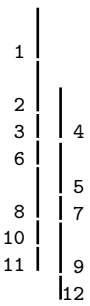
[9]In the following, we shall abusively confuse the table size and the least integer, parameter of the hash function.

one hashtable per dynamic type should be preferred. A first solution amounts to inlining the hashtable in the method table of the dynamic type and beginning with a cast to this dynamic type, which adds the following 3-instructions and $2L + 1$ cycles at the beginning of the sequence:

```
load [object + #tableOffset], table
load [table + #dcastOffset], delta
add object, delta, object
```

`DcastOffset` is the offset of the shift to the dynamic type subobject. This does not exploit possible parallelism and the total cycle number is $5L+6$. A better solution is as follows. The single hashtable is no longer inlined in a method table, but pointed by all of them, and each method table also contains the H_C parameter—this will allow some parallelism:

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #targetId, h, h
load [table + #htAddress], htable
load [table + #dcastOffset], delta1
add htable, h, htable
add object, delta1, object
load htable, id
load [htable + #2], delta2
comp #targetId, id
bne #fail
add object, delta2, object
```



$4L + 3$

`HtAddress` is the offset of the hashtable address—it replaces `htOffset` but cannot be avoided. The resulting code has the same length (12 instructions), with only $4L + 3$ cycles in case of success, the same as with one hashtable per method table (with $L = 2$). Note that the cost of dynamic hashing (lines 2-3) is only one cycle, since it is done in parallel with pointer adjustment.

3.4 Application to Java

Java and DotNet languages have a key particularity—i.e. classes are in single inheritance and interfaces in multiple subtyping. Dynamic types are always classes. As classes and interfaces are altogether types, subtyping test and method invocation differ according to whether the concerned type is a class or an interface.

3.4.1 *Subtype test.* When the target type is a class, Cohen's technique applies and there is no need for perfect hashing, which is more expensive. When the target type is an interface, perfect hashing applies, in a simplified form, compared to C++, since there is no shift (i.e. the position invariant holds).

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #targetId, h, hh
sub table, hh, table
load table, id
comp #targetId, id
```

$3L + 4$

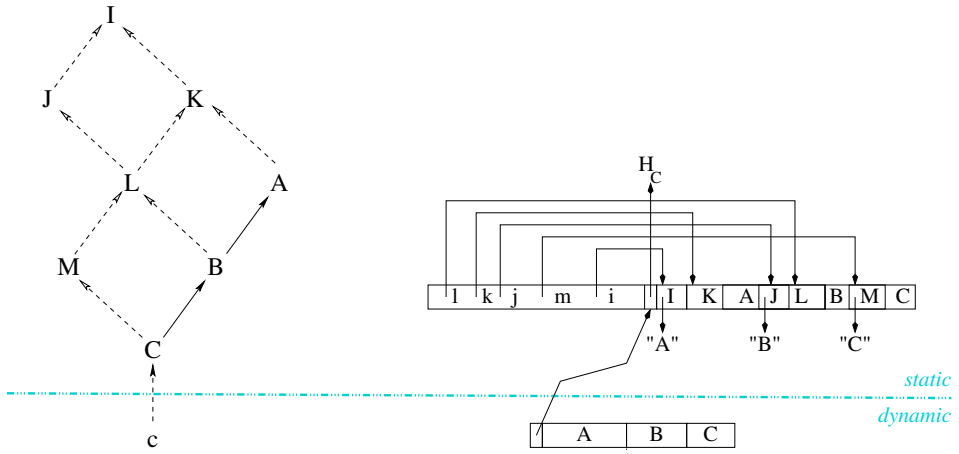


Fig. 4. A JAVA like, single inheritance and multiple subtyping hierarchy, with 3 classes A, B, C, 5 interfaces, I, J, K, L, M and an instance c. Object layout is the same as in single inheritance. The method table is twofold. On positive offsets is the method table itself, organized as with single inheritance. Negative offsets consist of the hashtable, which points, for each implemented interface, to the group of methods introduced by the interface

```
bne #fail
// success
```

In terms of cycles, this is only less than two times slower than Cohen's test. Regarding space, the size is divided by two, since each entry takes 2 bytes. Moreover, interfaces are fewer than classes and hashtables contain only interface identifiers. As there is no longer any pointer adjustment, the extra cost of dynamic hashing (lines 2-3) is $L + 1$ cycles.

Finally, when the target type is not statically known, a function must be called, which applies the technique corresponding to the kind of target type.

3.4.2 *Method invocation.* When the receiver is typed by a class, method invocation is the same as in single inheritance. Perfect hashing might be used for method invocation when the receiver is typed by an interface—i.e. the JVM `invokeinterface` operation [Meyer and Downing 1997]. In this case, the position invariant does not hold and there are at least two solutions, i.e. hashing interface or method identifiers.

3.4.2.1 *Interface method tables.* A first solution is a hashtable associating, with each interface implemented by a class, its own method table that contains only methods introduced by the interface. Rather than an extra method table, this is merely the address of the group of methods introduced by the interface within the class method table—so no extra space is needed (Figure 4). In this particular case, static typing makes the test useless since the search for the interface is always successful. Therefore, the hashtable contains only pointers at interface method tables and the code is even simpler:

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
```

```
and #id_C, h, h
sub htable, h, htable                                   4L + B + 2
load htable, itable
load [itable + #methodOffset], method
call method
```

The cycle number is $4L + B + 2$—where B is the *mispredicted branching latency* which may be greater than 10—while single inheritance method invocation takes only $2L + B$. Extra instructions w.t.t. single inheritance are italicized. Note that the two extra `load`s do not entail any cache miss risk, as the loaded address is in the already loaded method table.

Including all methods inherited by the interface in these interface tables would increase the size—specific tables would be needed in some cases, e.g. L in Figure 4—without substantially improving the time efficiency. Indeed, the only gain would be to cache these method tables, between two usages. However, as method invocation requires to save the context, i.e. to push this cache on the stack, this would only exchange, in some rare cases, 4 `load`s against one `store` and one `load`. An actual gain would suppose inlining the callee, but late binding makes inlining difficult [Detlefs and Agesen 1999]. Note also that the same hashtable should serve for both mechanisms since only one table can be inlined at negative offsets in the method tables—they are of variable length and `htOffset` must be a constant. Merging them is easy as they have the same keys and the first one only requires class identifiers, whereas the second only requires addresses. As both tables have word and half-word entries respectively, it would paradoxically increase the total size if word-alignment is required for efficiency. However, replacing addresses by offsets in the method table is a simple way to save space. Half-words are enough, at the expense of an extra `add`, between lines 5-6.

On the other hand, the part needed for Cohen's test, when the target type is a class, can be inlined in the positive offsets with method addresses.

3.4.2.2 *Hashing method identifiers.* An alternative is to hash method identifiers instead of interface identifiers. This implies that method identifiers are generated at load time, in the same way as class identifiers, by a simple numbering scheme.

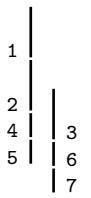
```
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #id_method, h, h                                    3L + B + 2
sub table, h, table
load table, method
call method
```

The advantage is to save one `load` (line 7-8 in the previous code) to the detriment of space, since a second hashtable must be used, as the sets of keys differ. Moreover, this second hashtable is larger since all methods introduced by interfaces must be hashed. Since there is no way to inline two hashtables in negative offsets, both hashtables must be merged—i.e. method and interface identifiers must differ, e.g. respectively even and odd, or with a single global numbering. In this case, there is only one entry per interface (resp. method), its identifier (resp. address), since method invocation is type safe.

3.4.3 *Direct access to interfaces.* In Section 2.2.1, we have ruled out direct access tables as they are too large—the space associated with a class C would be linear in the total number of classes (N) instead of the number n_C of superclasses of C. However, with single inheritance and multiple subtyping, there is only a need for direct access to interfaces. The total space would be $N_c \times N_i$, instead of $N \times N$, where $N = N_c + N_i$ and N_i, the number of interfaces, is presumed to be significantly less than N_c the number of classes. Therefore, a precise assessment requires an examination of direct access.

As for hashtables, the direct access array is inlined at negative offsets, starting from 0 in order to avoid one instruction. Each entry is either empty, or the address of the interface method table. The array size, i.e. the maximum identifier of implemented interfaces must also be stored in the method table. The subtyping test code is:

```
load [object + #tableOffset], table
load [table + #maxidOffset], max
load [table - #targetId], itable
comp #targetId, max
bgt #fail
comp itable, #empty
beq #fail
// succeed
```



$2L + 3$

With 7 instructions and $2L + 3$ cycles, this is as efficient as single inheritance techniques. Regarding method invocation, the following code is obtained:

```
load [object + #tableOffset], table
load [table - #targetId], itable
load [itable + #methodOffset], method
call method
```

$3L + B$

Once again, $3L + B$ cycles and 4 instructions are very efficient. As for interface hashing, addresses can be replaced by offsets, dividing the space by two, and adding only one cycle and one instruction to method invocation—the subtyping test is unchanged.

Another optimization would be to truncate the array to the minimum interface identifier implemented by the class. This implies storing the minimum and adding one addition, together with one test in the subtyping case. On the whole, this would give 11 instructions and $3L+3$ cycles in the subtyping code, and 6 instructions and $4L + B + 1$ cycles for method invocation.

3.5 Perfect hashing variants

Perfect hashing with `mod` or `and` functions is surely optimal from the standpoint of code length and from the standpoint of cycle number, but only in the case of `and`. However, one should expect that this is no longer optimal from the standpoint of table size. Therefore, one must also examine alternatives which would reduce the table size, presumably to the detriment of time and code length.

In the literature on perfect hashing [Czech et al. 1997], perfect hashing functions usually have more than one parameter. Adding a parameter would add an extra `load`—which would run in parallel, hence without extra cycle—plus an extra op-

eration. If this is a 1-cyle operation, the time overhead would be low. However, a second `mod`—this gives a classic perfect hashing function []—would markedly degrade the efficiency.

Another alternative is *quasi-perfect hashing* [Czech 1998]. Whereas perfect hashing consists in a function h which is injective on the considered set of keys I, quasi-perfect hashing involves a pair of hashing functions h_1 and h_2, such that if a key $x \in I$ is not found at place $h_1(x)$ in the first table, it will be found unambiguously at place $h_2(x)$ in the second table. A variant consists in a single function $h : \mathbf{N} \to [0..H-1]$ such that, $\forall x \in [0..H-1]$, $h^{-1}(x) \cap I$ has at most 2 elements. In that case, $h_1 = h$, $h_2 = h + H$ and the total size is $2H$.

Quasi-perfect hashing must be slightly adapted to the `and` case, when $h_C(x) = \texttt{and}(x, H_C)$, because the table size is $H_C + 1$, not H_C, and that we do not want to add $H_C + 1$ but only H_C. A key x is first searched at position $h_C(x)$ then at position $h_C(x) + H_C$, and the resulting table has size $2H_C + 1$. H_C is defined as the least integer such that all keys in I_C can be placed in the table without any collision.

The code for JAVA is transformed in the following way:

```
    load [object + #tableOffset], table
    load [table + #hashingOffset], h
    and #targetId, h, hh                          1
    sub table, hh, table
    load table, id                                2
    comp #targetId, id                            3
    beq #succeed                                  4                  3L + 5
    sub table, h, table                                 8
    load table, id                                5
    comp #targetId, id                            6     9
    bne #fail                                     7     10
                                                        11
succeed:    // success
```

A 4-instructions sequence is added (in italics) but it can be run in parallel and adds—theoretically—only one cycle.

This is to compare with an hypothetical two parameters hashing function, which might yield the following code, which has the same cycle number but 2 instructions less and only one test, i.e. one misprediction risk.

```
    load [object + #tableOffset], table
    load [table + #hashingOffset1], h1            1
    op1 #targetId, h1, hh                                            3L + 5
    load [table + #hashingOffset2], h2            2     4
    op2 hh, h2, hh                                3     5
    sub table, hh, table                                6
    load table, id
    comp #targetId, id                            7
    bne #fail                                     8
    // success                                    9
```

Table II. Implementations of subtype test: superclass number (n_C), bidirectional class coloring (COL_2), perfect hashing (PH) and quasi-perfect hashing—average and maximum per class—compared to class number (N)

name	n_C		COL_2		PH (`mod`)		PH (`and`)		qPH (`and`)		N
MI-jdk1.3.1	4.4	24	4.9	24	8.2	61	51.5	4100	32.3	8195	7401
MI-Orbix	2.8	13	2.8	13	4.4	32	12.3	2050	16.2	4099	2716
MI-Orbacus	4.5	19	4.7	19	8.1	50	24.0	1026	22.2	2051	1379
IBM-XML	4.4	14	4.6	14	7.6	43	21.2	148	28.7	259	145
IBM-SF	9.2	30	10.8	30	22.9	128	95.9	8224	36.7	4097	8793
MI-HotJava	5.1	23	7.4	23	9.8	85	32.8	519	23.9	1027	736
MI-Corba	3.9	18	4.2	18	6.5	48	19.5	1026	16.9	2051	1699
JDK.1.0.2	4.6	14	4.7	14	8.4	39	34.2	580	25.3	515	604
SmartEiffel	8.6	14	8.6	14	16.3	37	40.0	272	20.8	263	397
Unidraw	4.0	10	4.0	10	6.8	24	17.0	514	18.2	1027	614
Lov-obj-ed	8.5	24	11.4	24	19.4	90	53.2	508	30.1	513	436
Geode	14.0	50	17.4	50	41.8	267	160.8	1026	96.6	2051	1318

4. EXPERIMENTS AND STATISTICS

We have experimented perfect hashing on several large benchmarks commonly used in the object-oriented community. Algorithms for computing H_C values are presented in an Appendix. They are straightforward and the computation time is, on all benchmarks, quite insignificant. Class identifiers are generated by an arbitrary numbering of classes, in a possible class loading ordering—i.e. a *linear extension* (aka *topological sorting*) of $\prec$.

4.1 Benchmark description

Some large benchmarks are commonly used in the object-oriented implementation community[10], e.g. by [Vitek et al. 1997; Zibin and Gil 2001].

The benchmarks are abstract schemas of large class libraries, from various languages: Java (from MI-jdk1.3.1 to JDK.1.0.2 in Table II), Cecil (Cecil, Vortex3), Dylan, Clos (Harlequin), Self, Eiffel (SmartEiffel, Lov and Geode). Here, all benchmarks are multiple inheritance hierarchies and, in Java benchmarks, there is no distinction between classes and interfaces.

We recently developed an experimental platform—written in Common Lisp and Clos—for simulating and comparing various implementation techniques by computing spatial parameters [Ducournau 2005; 2006]. The following statistics result from a straightforward extension of this platform.

4.2 Application to plain multiple inheritance

Table II first presents two objective data—i.e. the number n_C of superclasses of each class C, including itself, (first columns) and the number of classes N (last column). Note that N is always far from 2^{15}—this confirms that short integers are suitable as class identifiers. $\Sigma_C n_C$ is the size of the transitive closure of the class hierarchy, i.e. a lower bound for all techniques derived from Cohen's test and the ideal linear-space size. The Table thus presents the result of class coloring according to the bidirectional variant. As coloring is a heuristics, these results may slightly

[10]Many people contributed to these benchmarks, including Karel Driesen and Jan Vitek: a current repository is Yoav Zibin's web site, `http://www.cs.technion.ac.il/~zyoav/`.

Table III. Number of introduced and inherited methods, size of method tables with coloring and standard multiple inheritance (average and maximum per class). The last columns present the ratio of the size of the method tables w.r.t. subtyping test, in the case of coloring and for subobject-based implementation, with modulus and bit-wise `and`

	methods				method tables				ratio			
name	introduced		inherited		COL_2		SMI		COL_2	`mod`	`and`	qPH
MI-jdk1.3.1	1.3	149	19.2	243	21.0	243	72.4	1391	8.7	8.9	1.4	2.2
MI-Orbix	0.4	64	8.3	109	8.6	109	23.0	534	6.1	5.3	1.9	1.4
MI-Orbacus	1.2	74	18.0	137	18.3	137	68.3	761	7.7	8.4	2.8	3.1
IBM-XML	2.5	29	16.1	57	16.3	57	50.4	284	7.1	6.6	2.4	1.8
IBM-SF	2.8	257	44.9	346	51.0	346	231.3	2063	9.4	10.1	2.4	6.3
MI-HotJava	1.8	80	34.2	189	37.7	189	134.8	817	10.1	13.7	4.1	5.6
MI-Corba	0.4	43	8.0	67	8.2	68	26.9	427	3.9	4.1	1.4	1.6
JDK.1.0.2	5.3	75	37.0	158	37.0	158	127.3	691	15.9	15.1	3.7	5.0
SmartEiffel	12.2	222	135.3	324	135.4	324	743.5	1576	31.3	45.5	18.6	35.8
Unidraw	2.9	103	24.1	124	24.1	124	68.9	318	12.0	10.1	4.1	3.8
Lov-obj-ed	8.3	117	85.9	289	113.5	289	422.1	1590	19.8	21.8	7.9	14.0
Geode	6.1	193	231.8	880	291.0	892	1445.6	10717	33.5	34.6	9.0	15.0

differ from comparable ones. Finally, results of perfect hashing (PH) are presented, with modulus and bit-wise `and` hashing functions. These results are produced by exact algorithms, so they should be exactly reproducible.

Note first that the coloring results are very close to n_C, i.e. Cohen's test. In a framework of global linking or global compilation, the overhead is not significant compared to subobject-based implementations.

Modulus based perfect hashing is not as good as coloring, but it is almost as good as a good linear probe. On average, H_C is always smaller than $3n_C$ and smaller than 2 fold the coloring. As a good linear probe parameter should be close to $2n_C$, the overhead of perfection—i.e. the overhead of perfect hashing w.r.t. linear probong—is not significant. Note however that hashtable entries are double, hence the complete size will be 4-fold the coloring.

With bit-wise `and`, the result is not so good—H_C is between 2- and 6-fold greater than with modulus. On the other hand, the H_C average remains between 7- and 240-fold smaller than the class number N, whereas $N/2$ would be the average size of direct access tables. In all benchmarks but the most demanding, Geode, H_C is far lower than $\sqrt{n_C N}$.

Quasi-perfect hashing (qPH) is better than PH `and` on average and for most benchmarks, except MI-orbix, IBM-XML and Unidraw. However, on all benchmarks, PH `mod` remains far better.

Finally, Table III presents statistical data on methods (introduced or inherited), size of method tables with coloring and subobject-based implementation. Note that the number of inherited methods gives the exact size of method tables in single inheritance. The Table ends with ratios between the size of method tables and subtype tables, in the coloring and subobjects frameworks. This ratio is always greater than 4 for coloring and modulus PH, but it may be quite larger (up to 40 with coloring and 70 for modulus). In contrast, the ratio with bit-wise `and` is not so high. However, it is always greater than 4 with plain multiple inheritance—i.e. only Java benchmarks have a smaller ratio, but the approach would only be used in Java when the target type is an interface. Furthermore, the overhead of perfection is also far lower than that of subobject-based implementations w.r.t.

Table IV. Classes and interfaces in JAVA benchmarks—total number of classes and interfaces, number of implemented interfaces and extended classes per class (average and maximum per class), and average size of direct access (DA and DAt) tables

name	class	interface	implemented		extended		DA	DAt
MI-jdk1.3.1	7056	345	1.2	21	3.2	8	39.9	23.0
MI-Orbix	2676	40	0.3	8	2.4	8	1.8	0.9
MI-Orbacus	1297	82	1.7	14	2.9	7	10.6	9.0
IBM-XML	129	16	1.7	9	2.9	6	3.6	2.4
IBM-SF	7920	873	6.2	27	2.8	9	283.2	270.8
MI-HotJava	681	55	2.4	19	2.9	7	10.6	9.2
MI-Corba	1634	65	1.1	14	2.7	6	6.0	4.6
JDK.1.0.2	576	28	1.1	9	3.6	8	6.4	3.9

single inheritance.

4.3 Application to JAVA

As classes and interfaces are not distinguished in the benchmarks that we used, we computed interfaces according to a heuristics which should give a good approximation [Ducournau 2005, Appendix A]. Table IV first presents the number of classes and interfaces, along with the average size of the direct access bit-array (DA) with its truncated variant (DAt). Truncation obviously has a minor effect and we shall no longer consider it.

Hashing of interfaces. Table V presents the results of perfect hashing of interfaces, and the respective sizes of the positive and negative parts of the method table. As perfect hashing has better result when the identifiers are small, used interface identifiers result from a specific numbering of interfaces. The positive part contains method addresses and class identifiers for the Cohen's test, with 2 identifiers per word. The negative part only contains the interface hashtable, with two words per entry—i.e. one assumes that word-alignment is required. Remember that the space could be divided by two, if addresses are replaced by offsets, at the expense of one extra cycle. Implemented interfaces give the current n_C parameter and perfect hashing H_C parameters follow.

Modulus still gives very good results, always less than $2n_C$, except for IBM-SF, which is slightly more. Modulus perfect hashing compares favourably with direct access tables, since the DA/PH ratio is, on average, greater than 10. Of course, this large ratio is due to the largest benchmarks, IBM-SF and JDK1.3.1.

Bit-wise `and` is still inferior, but acceptable. Table V also presents the space occupied by positive offsets—i.e. method table itself and Cohen's test (left columns) and the ratio between negative and positive parts. Note that the method numbers differ from the corresponding ones in Table III, since interfaces are removed, but they differ slightly, as interfaces are removed from both numerator (i.e. table size) and denominator (i.e. number), and there are only a few of them. These ratios are always less than 1—i.e. the interface part is smaller than the class part—except in the case of IBM-SF, the largest and most demanding benchmark, with bit-wise `and`. Clearly, the cost of modulus-based perfect hashing is low and bit-wise `and` is higher, but it is surely worthwhile to save on a few cycles, since subtyping tests and `invokeinterface` are extensively used in JAVA and DOTNET languages.

Table V. Perfect hashing of JAVA interfaces—positive size (extended classes for Cohen's test, and methods), implemented interfaces to be hashed per class (i.e. the current parameter n_C), perfect hashing parameters (PH) and ratio between negative and positive size, for both direct access and perfect hashing (average per class)

	pos. offsets			neg. off.	PH			neg/pos			
name	class	meth.	tot.	inter.	`mod`	`and`	qPH	`mod`	`and`	qPH	DA
MI-jdk1.3.1	3.2	19.2	20.8	1.2	1.9	4.0	3.2	0.2	0.4	0.3	1.9
MI-Orbix	2.4	8.2	9.4	0.3	0.4	1.4	1.4	0.1	0.3	0.3	0.2
MI-Orbacus	2.9	17.9	19.3	1.7	2.7	4.1	3.6	0.3	0.4	0.4	0.5
IBM-XML	2.9	16.4	17.8	1.7	2.3	3.4	3.5	0.3	0.4	0.4	0.2
IBM-SF	2.8	44.0	45.4	6.2	13.5	40.4	21.9	0.6	1.8	1.0	6.2
MI-HotJava	2.9	34.7	36.2	2.4	3.9	5.9	5.5	0.2	0.3	0.3	0.3
MI-Corba	2.7	7.7	9.1	1.1	1.6	3.0	2.7	0.4	0.7	0.6	0.7
JDK.1.0.2	3.6	37.5	39.3	1.1	1.4	2.1	2.0	0.1	0.1	0.1	0.2
Total	2.9	26.5	28.0	3.0	6.0	16.7	9.7	0.4	1.2	0.7	4.2

Table VI. Perfect hashing of JAVA interfaces and methods—number of interfaces and methods to be hashed per class (i.e. the sum is the current parameter n_C), perfect hashing parameters and negative vs. positive size ratio (average per class)

	pos. off.	neg. offsets			PH			neg/pos		
name	tot.	inter.	meth.	tot.	`mod`	`and`	qPH	`mod`	`and`	qPH
MI-jdk1.3.1	20.8	1.2	8.1	9.3	23.1	87.9	27.7	1.1	4.2	1.3
MI-Orbix	9.4	0.3	1.3	1.7	2.1	5.1	4.3	0.2	0.5	0.5
MI-Orbacus	19.3	1.7	8.7	10.4	19.6	44.3	28.3	1.0	2.3	1.5
IBM-XML	17.8	1.7	8.8	10.5	19.6	25.7	20.7	1.1	1.4	1.2
IBM-SF	45.4	6.2	31.6	37.7	117.1	558.5	182.2	2.6	12.3	4.0
MI-HotJava	36.2	2.4	20.9	23.3	47.4	64.1	57.6	1.3	1.8	1.6
MI-Corba	9.1	1.1	3.2	4.3	6.4	14.4	9.9	0.7	1.6	1.1
JDK.1.0.2	39.3	1.1	3.1	4.2	5.4	33.1	10.8	0.1	0.8	0.3
Total	28.0	3.0	15.7	18.6	53.3	236.9	79.7	1.9	8.5	2.8

Hashing of methods and interfaces. Table VI gives the results of perfect hashing of methods and interfaces. Identifiers are the result of a single numbering of both interfaces and methods—actually two distinct numbering schemes, respectively odd and even, gave bad results. The negative part contains the same column for extended interfaces, plus the number per class of inherited methods introduced by an interface. The sum is the n_C parameter. The modulus results are not as good as previously—here H_C is greater than $2n_C$ in several benchmarks—and bit-wise `and` is even worse. Compared to hashing of interfaces, hashing of methods requires about 10-fold more entries, but they are single entries. With bit-wise `and`, the negative part may be more than 10-fold larger than the positive part, i.e. single inheritance method tables. This is more than the overhead of subobject-based implementation vs. coloring (Table III).

Conclusions. Four variants of perfect hashing are to compare, according to what is hashed—interfaces or methods—and which hashing function is used—modulus or bit-wise `and`. Direct access to interfaces must also be considered, together with its truncated variant. Moreover, in the case of interfaces (hashing or direct access), one may also distinguish between addresses and offsets, with the latter dividing the size by two but adding one cycle and one instruction. Note that we did not measure direct acces to methods, as the required space is unreasonable. This gives the nine lines of Table VII, ordered by the cycle number of method invocation—class coloring

Table VII. Comparison of all techniques, ordered by cycle numbers for method call (with $L = 3$ and $B = 10$). The last column indicates the techniques which may be retained (+) and those which must be ruled out (–), since they are surpassed by another one w.r.t. all criteria.

technique			method call cycles	method call code	subtype test cycles	subtype test code	space neg/pos	
class coloring			$2L + B = 16$	3	$2L + 2 = 8$	4	–	
DA	interface	addr.	$3L + B = 19$	4	$2L + 3 = 9$	7	4.2	+
DA	interface	offset	$3L + B + 1 = 20$	5	$2L + 3 = 9$	7	2.1	+
PH `and`	method		$3L + B + 2 = 21$	6	$3L + 4 = 13$	7	8.5	–
PH `and`	interface	addr.	$4L + B + 2 = 24$	7	$3L + 4 = 13$	7	1.2	+
qPH `and`	interface	addr.	$4L + B + 3 = 25$	11	$3L + 5 = 14$	11	0.7	–
PH `and`	interface	offsets	$4L + B + 3 = 25$	8	$3L + 4 = 13$	7	0.6	+
qPH `and`	interface	offsets	$4L + B + 4 = 26$	12	$3L + 5 = 14$	11	0.4	+
PH `mod`	method		$3L + B + 7 = 26$	6	$3L + 9 = 18$	7	1.9	–
PH `mod`	interface	addr.	$4L + B + 7 = 29$	7	$3L + 9 = 18$	7	0.4	+
PH `mod`	interface	offsets	$4L + B + 8 = 30$	8	$3L + 9 = 18$	7	0.2	+

has been added for comparison.

A first observation is that method perfect hashing is ruled out. Direct access is better than bit-wise `and` method perfect hashing on all criteria, whereas interface hashing with bit-wise `and` compares favourably with modulus method perfect hashing. From a time standpoint, modulus adds 5 cycles (integer division), while interface hashing adds L cycles—i.e. one `load`, without any cache miss chance, since the interface method table is part of the class method table, which is already in cache. So interface hashing with bit-wise `and` should be slightly faster, as long as `load` is faster than integer division. From the space standpoint, it is also smaller for almost all benchmarks. This rules out modulus method hashing.

Regarding truncated direct access, the space gain upon direct access is too small and direct access to interface offsets is clearly better.

Note that, by no means, shorter code sequences compensate for larger tables. The number of sites of interface typed method invocations is rather low, certainly not much more than a few per class. [Ducournau 2005] reports an averagee of less than 10 method definitions per class (including method declaration in interfaces) and one of the few available statistics on the number of call sites [Driesen et al. 1995] indicates an average of 4 call sites per method definition. Only a few of them may be presumed to be typed by an interface. On the whole, the single instruction saved on by method perfect hashing cannot compensate for the 200 extra words per class entailed by bit-wise `and`.

Generally, three variant groups remain, from the fastest and largest to the slowest and smallest. Direct access to interfaces may be the solution if time-efficiency has priority and if 5-fold space increasing is accepted—however, one must not forget its bad scalability, as its space is directly related to the number of interfaces. On the contrary, modulus interface perfect hashing is the solution if space is the priority, as the space increase is less than 50%. Finally, bit-wise `and` interface perfect hashing is a midway solution.

As these variants are very close to each other in terms of computation, they could easily be a parameter of the run-time system. Moreover, as modulus latency depends on processors, the conclusion must take the processor characteristics into account.

5. RELATED WORKS

Many people have contributed to subtyping tests. This paper focuses on techniques inherited from [Cohen 1991], as they are efficient in practice. A rather different approach is based on mathematical encoding of partial orders [Aït-Kaci et al. 1989; Caseau 1993; Habib and Nourine 1994; Habib et al. 1995; Habib et al. 1997; Raynaud and Thierry 2001]. The objective is then optimal encoding, thus minimizing the size. [Fall 1998] surveyed this approach, which is based on mathematical properties of posets like dimension or products. However, a minimal size may be unrealistic, in practice, as the code for extracting the data becomes quite intricate. A good example is given by *bit pack encoding* [Vitek et al. 1997] which is a variant of *pack encoding*—i.e. class coloring—where the class identifiers are no longer global but depend on the class color and the size of the field associated with a color depends on the number of classes with the same color. The resulting tables are markedly smaller, but the decoding code is markedly longer. Moreover, these optimized encodings are uncompatible with dynamic loading, as they are not incremental and their complexity is too high, i.e. often not-polynomial.

Among the various recent techniques, Zibin and Gil [2001] propose a combination of Schubert's numbering and class coloring, called *PQ-encoding*. It probably gives the most compact constant-time technique available for multiple inheritance, but it is not compatible with dynamic loading. Moreover, as the encoding is complex, there is no way to recompute it at each class loading.

Cohen's test is commonly used in the implementation of Java virtual machines for class subtyping tests [Alpern et al. 2001; Click and Rose 2002]. However, it is often used in a non-optimal way, with fixed-length arrays in order to avoid array bound checking and without inlining this array in the method table, hence with one extra `load`. Regarding interface subtyping tests, [Alpern et al. 2001] uses a direct access array indexed by interface identifiers. This is a more acceptable solution than in the subobject-based framework, since this may be a boolean array (no shift), with dimension $N_c \times N_i$, instead of $N \times N$—where N_c and N_i are the number of classes and interfaces. However, it has some drawbacks: it needs array bound checking, extra code for bit-extraction—unless it is a byte-array—and it does not work for interface method invocation. Some other experimental implementations, e.g. [Krall and Grafl 1997; Gagnon and Hendren 2001], use direct access—our simulation shows that direct access to interfaces is not excessively space-consuming. In contrast, direct access to methods is over space-consuming and [Gagnon and Hendren 2001] allocates objects in the empty area of these huge tables.

In contrast, [Click and Rose 2002] uses a simple linear search but improves it with a cache in the source type memorizing the last target type for which the test succeeded. Of course this improvement might serve for any table-based subtyping technique. In the best cases, this cache allows a 2-`load` test, instead of 3-`load` with perfect hashing[11]. However, cache misses would entail 4-`load` tests. Obviously, the improvement is a matter of statistics. Regarding space, the authors report 8–11 words per class. This is larger than modulus interface perfect hashing, except for IBM-SF which seems to be larger than the hierarchy used for Click and Rose

[11] We count here the `load` access to the method table from the object itself, whereas the authors consider only the `load` from the method table, i.e. the class structure.

tests—indeed, they report a maximal depth of always less than 8, whereas IBM-SF reaches 9 (Table IV).

[Alpern et al. 2001] reviews interface typed method invocation in Java and Dot-Net languages. One basic approach is a linear search of the considered interface in the class method table. In Jalapeño, [Alpern et al. 2001] associates a fixed-size hashtable with each class, mapping each method identifier to the address of a method or of a decision tree indexed on method identifiers. It could be considered as a perfect hashtable—but the value associated with an entry may be a dispatch tree—or a classic hashtable using separate chaining, where the collision resolution is no longer intepreted, but compiled instead. The advantage is that hashing is uniformly and statically done—no `load` or hash function at run-time and a compiled collision resolution chaining is likely more efficient that interpreted linear probing. The drawbacks are unconstant-time—though the `load` number is constant and small—together with the fact that it does not work with subtyping tests. This paper presents benchmarks with two values, 5 and 40, for the H parameter. The former is in the same region as the average H_C in modulus interface perfect hashing—i.e. with the same space cost, this technique replaces dynamic hashing by compiled probes (Table VI). The latter is much greater than interface perfect hashing, and even greater than direct access, except in the IBM-SF case (Table V). From both space and time standpoints, direct access should be better on most benchmarks.

Bidirectionality of method tables is become common in virtual machines, at least for experimental ones—e.g. [Krall and Grafl 1997; Gagnon and Hendren 2001]—and derives from [Pugh and Weddell 1990; Myers 1995].

On the other hand, not all object-oriented implementations are based on method tables. In the GNU Eiffel compiler, Smart Eiffel, method tables are not used. Instead, objects are tagged by their class identifier and all polymorphic operations—particularly subtyping tests—are implemented using small dispatch trees [Zendra et al. 1997; Collin et al. 1997]. However, the approach is practical only because compilation is global, hence all classes are statically known. Furthermore, a type analysis is used to restrict the set of *concrete types* [Bacon and Sweeney 1996; Gil and Itai 1998; Grove and Chambers 2001] and make dispatch efficient.

Perfect hashing has been used for closed symbol tables, e.g. reserved words [Schmidt 1990], or for constant-time access to sparse arrays. There is little evidence of its usage in an object-oriented framework, especially when applied to pure object-oriented mechanisms. Driesen [1993a], Zibin and Gil [2003] discard it for method dispatch, as they consider that it is too complicated, with a too high constant, or a too high complexity—actually, they consider *minimal* perfect hashing functions, i.e. functions such that $H_C = n_C$. However, sparse table compression can be considered as a special case of perfect hashing [Czech et al. 1997]. Besides coloring, this includes *row displacement* [Tarjan and Yao 1979] which has been applied to method invocation by [Driesen 1993b; Driesen and Hölzle 1995; Driesen 2001]. Row displacement has not been explicitly proposed for subtype tests, though it would obviously apply, like coloring. However, row displacement is compatible with dynamic loading only in its *class-based* variant. Accroding to Driesen, this might be less space-efficient than the selector-based variant. In contrast, [Klefstad et al. 2002] uses perfect hashing for method dispatch in the framework of CORBA

distributed applications—method names are hashed, which is more expensive than hashing small integers.

Finally, to our knowledge, the various perfect hashing functions proposed in the literature do not meet our requirements. For instance, *quotient reduction* [Sprugnoli 1977] involves a function $x \mapsto \lfloor (x+s)/n \rfloor$, which presents the same disadvantage as `mod` augmented by an extra parameter, a need for a bound check and a very bad compacity with consecutive numbers.

6. CONCLUSION AND PERSPECTIVES

In this paper, we have proposed a subtyping test based on perfect hashing, which generalizes the test proposed by Cohen [1991] and class coloring [Vitek et al. 1997]. Indeed, class coloring may be understood as perfect hashing, where all hash functions are the coloring function—i.e. $h_C(D) = \chi(D)$.

The resulting subtype test satisfies our five initial requirements:

constant-time. the code sequence takes $3L+4$ cycles with bit-wise `and`, i.e. around 2-fold coloring; modulus would add around 5 cycles, maybe more;

linear-space. with modulus, the size of the tables dedicated to subtyping tests is about 4-fold coloring and always 4-fold smaller than method tables in subobject-based implementation; with bit-wise `and`, the ratio is not as good, but it is still 4-fold smaller than method tables if one rules out Java benchmarks;

multiple inheritance. perfect hashing works well with plain multiple inheritance and it may apply to Java in a more efficient way—i.e. restricted to interfaces and using hashsets instead of hashtables;

dynamic loading. the technique requires only class numbering at load or link time, together with the computation of H_C with straightforward algorithms;

inlining. the code sequence takes 12 instructions in a subobject-based implementation, and only 7 in a reference-invariant implementation.

Clearly, these results are not perfect—i.e. smaller, shorter, faster would be better. However, in the present state of affairs, this is a practical technique. The approach might be applied to C++ compilers, at two minor technical conditions: i) linkers should be adapted to class numbering and H_C computation; ii) perfect hashing should be adapted to *non-virtual inheritance*, an impure form of multiple inheritance used in this language when superclasses are not annotated by the `virtual` keyword [Ellis and Stroustrup 1990; Lippman 1996; Ducournau 2005].

The application to Java and DotNet languages is even more straightforward as their runtime systems are equipped with a loader. So perfect hashing may be used for subtyping tests when the target is an interface—with Cohen's test being preferred for class targets. Furthermore, perfect hashing also applies to method invocation, when the receiver is typed by an interface, and both mechanisms use the same hashtable. However, we have not considered, in our requirements, the case of unloading and reloading classes. This should not be a problem, at least as long as unloading a class entails unloading all its subclasses.

Chosing between the two hashing functions involves a tradeoff between time and space as bit-wise `and` takes about 5 cycles less than integer division. In the framework of plain multiple inheritance, the overall size of subtyping test tables is

substantially smaller than method tables, regardless of the function used. Moreover, in C++, the subtyping test is generally not critical. Therefore, modulus perfect hashing is a good compromise.

Application to JAVA should favour time, as the technique applies to both subtype test and method invocation. Moreover, subtyping checks are quite common in JAVA programs, due to the lack of genericity (up to version 1.5) and to covariant array subtyping. However, several efficient techniques compete—i.e. interface or method perfect hashing and direct access. Our simulation clearly shows that perfect hashing of methods—at least with the functions that we tested—compare badly with direct access to interfaces. Therefore, the choice is between direct access to interfaces—priority to time—or modulus perfect hashing—priority to space. Bit-wise `and` perfect hashing may be a good compromise.

Anyway, it would certainly be worthwhile to search for more efficient hash functions—we could not imagine another function than modulus and bit-wise `and`. Quasi-perfect hashing improves the size at the expense of a 2-probe test. A better solution might be a two-parameter function family, which would add 2 cycles and 2 instructions to bit-wise `and`, and should reduce the space to that of modulus, or even less.

Perfect hashing could also be applied to object layout—this might be understood as *incremental coloring*. In this case, reference invariant holds and the object layout is the same as with single inheritance, except that the position invariant does not hold. Instead, attributes are grouped according to the class which introduces them. The method table is the hashtable with 3-fold entries—i.e. class identifier, offset of the corresponding attribute group in the object layout[12] and a pointer to the method table grouping all methods introduced by the considered class. Overall, subtyping tests and method invocation would be the same as previously described for JAVA. On the other hand, accesses to attributes would require two indirections—this is considerable compared to single inheritance but almost the same as with subobject-based implementation, when the receiver's static type is not the class introducing the attribute. The double compilation approach proposed by Myers [1995] would not apply as it requires a global linking framework.

On the whole, perfect hashing is a 30-year old technique that we propose to apply to a 25-year old problem. There is no evidence that it has already been done but there is still room for doubt.

REFERENCES

AGRAWAL, R., BORGIDA, A., AND JAGADISH, H. 1989. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data, Portland (OR), USA*. ACM SIGMOD Record, 18(2). 253–262.

ALPERN, B., COCCHI, A., FINK, S., AND GROVE, D. 2001. Efficient implementation of Java interfaces: Invokeinterface considered harmless. See OOPSLA [2001].

ALPERN, B., COCCHI, A., AND GROVE, D. 2001. Dynamic type checking in Jalapeño. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*.

AÏT-KACI, H., BOYER, R., LINCOLN, P., AND NASR, R. 1989. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems 11,* 1, 115–146.

[12]This is called *accessor simulation* in [Ducournau 2005].

Bacon, D. and Sweeney, P. 1996. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*. SIGPLAN Notices, 31(10). ACM Press, 324–341.

Caseau, Y. 1993. Efficient handling of multiple inheritance hierarchies. In *Proc. OOPSLA'93*. SIGPLAN Notices, 28(10). ACM Press, 271–287.

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

Cohen, N. 1991. Type-extension type tests can be performed in constant time. *Programming languages and systems 13,* 4, 626–629.

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

Czech, Z. J. 1998. Quasi-perfect hashing. *The Computer Journal 41*, 416–421.

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

Detlefs, D. and Agesen, O. 1999. Inlining of virtual methods. In *Proc. ECOOP'99*, R. Guerraoui, Ed. LNCS 1628. Springer, 258–277.

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*. ACM Press.

Driesen, K. 1993a. Method lookup strategies in dynamically typed object-oriented programming languages.

Driesen, K. 1993b. Selector table indexing and sparse arrays. In *Proc. OOPSLA'93*. ACM Press.

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

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. See OOPSLA [1995], 141–155.

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.

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. 2002a. La coloration pour l'implémentation des langages à objets à typage statique. In *Actes LMO'2002* in *L'Objet vol. 8*, M. Dao and M. Huchard, Eds. Lavoisier, 79–98.

Ducournau, R. 2002b. "Real World" as an argument for covariant specialization in programming and modeling. In *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, J.-M. Bruel and Z. Bellahsène, Eds. LNCS 2426. Springer, 3–12.

Ducournau, R. 2005? Implementing statically typed object-oriented programming languages. *ACM Computing Surveys*. (submitted 2002; revised 2005).

Ducournau, R. 2006. Coloring, a versatile technique for implementing object-oriented languages. Tech. Rep. 06-001, L.I.R.M.M., Université Montpellier 2. (submitted).

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

Fall, A. 1998. The foundations of taxonomic encoding. *Computational Intelligence 14*, 598–642.

Gagnon, E. M. and Hendren, L. 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*.

Garey, M. and Johnson, D. 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.

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

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

Habib, M., Huchard, M., and Nourine, L. 1995. Embedding partially ordered sets into chain-products. In *Proc. KRUSE'95*, G. Ellis, R. A. Levinson, A. Fall, and V. Dalh, Eds. 147–161.

Habib, M. and Nourine, L. 1994. Bit-vector encoding for partially ordered sets. In *ORDAL*, V. Bouchitté and M. Morvan, Eds. Lecture Notes in Computer Science, vol. 831. Springer, 1–12.

Habib, M., Nourine, L., and Raynaud, O. 1997. A new lattice-based heuristic for taxonomy encoding. In *Proc. KRUSE'97*. 60–71.

Kiczales, G., des Rivières, J., and Bobrow, D. 1991. *The Art of the Meta-Object Protocol*. MIT Press.

Klefstad, R., Krishna, A., and Schmidt, D. 2002. Design and performance of a modular portable object adapter for distributed, real-time, and embedded corba applications. In *Proc. of the 4th International Symposium on Distributed Objects and Applications*. OMG.

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

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

Krall, A., Vitek, J., and Horspool, R. 1997. Near optimal hierarchical encoding of types. In *Proc. ECOOP'97*, M. Aksit and S. Matsuoka, Eds. LNCS 1241. Springer.

Lippman, S. 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.

Mehlhorn, K. and Tsakalidis, A. 1990. Data structures. See Van Leeuwen [1990], Chapter 6, 301–341.

Meyer, B. 1992. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK.

Meyer, B. 1997. *Object-Oriented Software Construction*, second ed. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA.

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

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

Myers, A. 1995. Bidirectional object layout for separate compilation. See OOPSLA [1995], 124–139.

OOPSLA 1995. *Proceedings of the Tenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'95*. SIGPLAN Notices, 30(10). ACM Press.

OOPSLA 1997. *Proceedings of the Twelfth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*. SIGPLAN Notices, 32(10). ACM Press.

OOPSLA 2001. *Proceedings of the Sixteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'01*. SIGPLAN Notices, 36(10). ACM Press.

Pfister, B. H. C. and Templ, J. 1991. Oberon technical notes. Tech. Rep. 156, Eidgenossische Techniscle Hochschule Zurich–Departement Informatik.

Pugh, W. and Weddell, G. 1990. Two-directional record layout for multiple inheritance. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'90)*. ACM SIGPLAN Notices, 25(6). 85–91.

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

Queinnec, C. 1997. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*.

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.

Schmidt, D. C. 1990. GPERF: A perfect hash function generator. In *USENIX C++ Conference*. 87–102.

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

Sprugnoli, R. 1977. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM 20,* 11 (November), 841–850.

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

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

Takhedmit, P. 2003. Coloration de classes et de propriétés : étude algorithmique et heuristique. M.S. thesis, Université Montpellier II.

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

Van Leeuwen, J., Ed. 1990. *Algorithms and Complexity.* Handbook of Theoretical Computer Science, vol. 1. Elsevier, Amsterdam.

Vitek, J., Horspool, R., and Krall, A. 1997. Efficient type inclusion tests. See OOPSLA [1997], 142–157.

Vitter, J. S. and Flajolet, P. 1990. Average-case analysis of algorithms and data structures. See Van Leeuwen [1990], Chapter 9, 431–524.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. See OOPSLA [1997], 125–141.

Zibin, Y. and Gil, J. 2001. Efficient subtyping tests with PQ-encoding. See OOPSLA [2001], 96–107.

Zibin, Y. and Gil, J. 2003. Incremental algorithms for dispatching in dynamically typed languages. In *Proc. of ACM Conf. on Principles of Programming Languages (POPL'03)*. 126–138.

Received December 2005

A. APPENDIX: ALGORITHMS

Subtype checks are presented in a simple COMMON LISP code [Steele 1990].

A.1 Modulus

The algorithm for computing H_C in the modulus case is quite straightforward. The entry is a list of integers, presumed to be the set of identifiers of C superclasses. Starting from n_C, the length of the list, each integer is tested to determine if it gives an injective hashing function h_C. Injectivity is checked by building a hashtable—an array—and hashing all numbers.

```
(defun perfect-hash-mod (ln)
  ;; ln is a list of positive integers
  (loop for n from (length ln) by 1
        until (perfect-hashing-p ln n #'(lambda (x) (mod x n)))
        finally return n))

(defun perfect-hashing-p (ln size fn)
  (let ((ht (make-array size)))
    (loop for i in ln
          finally return size
          do
          (let ((hv (funcall fn i)))
            (if (aref ht hv)
                (return nil)
              (setf (aref ht hv) i))))))
```

A.2 Bit-wise and

Bit-wise **and** is slightly more difficult. One first computes all discriminant bits, i.e. bits which are not 0 or 1 everywhere. The resulting integer gives a perfect hashing function since all integer pairs differ by at least one bit. Then one checks each 1-value bit, by decreasing weight, and one switchs the bit when it is not required for injectivity.

```
(defun perfect-hash-and (ln)
  ;; ln is a list of positive integers
  (if (null (cdr ln))
      1
    (let ((mask (logxor (apply #'logior ln)
                        (apply #'logand ln))))
      ;; mask contains only discriminant 1-bits
      (check-one-bit mask ln (1- (integer-length mask)))
      )))

(defun check-one-bit (mask ln bit)
  (if (null bit)
      (1+ mask)
    (let ((new (logxor mask (ash 1 bit))))
      (cond ((= new 0) (1+ mask))
            ((perfect-hashing-p ln (1+ new) #'(lambda (x) (logand x new)))
             (check-one-bit new ln (next-bit new bit)))
```

```
            (t (check-one-bit mask ln (next-bit mask bit)))))))

(defun next-bit (mask bit)
  (loop for b from (1- bit) by 1 downto 0
        when (logbitp b mask) return b))
```

In the previous code, `logxor`, `logior` and `logand` are COMMON LISP integer functions for bit-wise operations: exclusive and inclusive `or`, and `and`. `Integer-length` gives the position of the first left 1-bit of a positive integer. `(logbitp n b)` tests if the `b`-th bit of `n` is 1. `(Ash n b)` shifts `n` left by `b` positions (when `b` is positive).