SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:661–694
Published online 28 December 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1024

Perfect class hashing and numbering for object-oriented implementation

Roland Ducournau*, † and Floréal Morandat

LIRMM—Université Montpellier II and CNRS, France

SUMMARY

Late binding and subtyping create run-time overhead for object-oriented languages, especially in the context of both *multiple inheritance* and *dynamic loading*, for instance for JAVA interfaces. In a previous article, we proposed a novel approach based on *perfect hashing* and truly constant-time hashtables for implementing subtype testing and method invocation in a dynamic loading setting. In this first study, we based our efficiency assessment on Driesen's abstract computational model for the time aspect, and on large-scale benchmarks for the space aspect. The conclusions were that the technique was promising but required further research in order to assess its scalability. This article presents some new results on perfect class hashing that enhance its interest. We propose and test both new hashing functions and an inverse problem that amounts to selecting the best class identifiers in order to minimize the overall hashtable size. This optimizing approach is proven to be optimal for single-inheritance hierarchies. Experiments within an extended testbed with random class loading and under cautious assumptions about what should be a sensible class-loading order show that perfect class hashing scales up gracefully, especially on JAVA-like multiple-subtyping hierarchies. Furthermore, perfect class hashing is implemented in the PRM compiler testbed, and compared here with the coloring technique, which amounts to maintaining the single-inheritance implementation in multiple inheritance. The overall conclusion is that the approach is efficient from both time and space standpoints with the bit-wise `and` hashing function. In contrast, the poor time efficiency of *modulus* hashing function on most processors is confirmed. Copyright © 2010 John Wiley & Sons, Ltd.

Received 16 February 2010; Revised 6 September 2010; Accepted 22 September 2010

KEY WORDS: object-oriented programming; dynamic loading; multiple inheritance; late binding; subtype test; perfect hashing

1. INTRODUCTION

The implementation of object-oriented languages is an important issue in the context of both *multiple inheritance* and *dynamic loading*. Object-oriented implementation concerns a few basic mechanisms that are invoked billions of times during a 1-minute program execution. Very high efficiency is necessary to ensure this intensive use, and it is arguable that all implementations of object-oriented languages, especially in this context, should fulfill the following requirements: (i) *constant-time*, (ii) *inlining* and (iii) *linear-space*. Constant time is the only way to bound the worst-case behavior. Whereas it is essential in a real-time setting, it remains highly preferable in all settings. Moreover, the basic mechanisms must be implemented by a short sequence of instructions that is inlined, thus saving a function call and reducing the time constant. Finally, space linearity ensures that the implementation will scale up gracefully with the program size; we

*Correspondence to: Roland Ducournau, LIRMM—Université Montpellier II and CNRS, 161 rue Ada, 34000 Montpellier, France.
†E-mail: ducour@lirmm.fr, Roland.Ducournau@lirmm.fr

shall see, however, that linearity must be understood with a slightly specific meaning. This general implementation issue is exemplified by the two most used languages that support both features, namely C++ and JAVA. When the `virtual` keyword is used for inheritance, C++ provides a fully reusable implementation, based on subobjects, which however implies a lot of compiler-generated fields in the object layout and pointer adjustments at run-time[‡] [1]. Moreover, it does not meet the linear-space requirement and, until recently, there were no known efficient subtype tests available for this implementation. JAVA provides multiple inheritance of interfaces only but, even in this restricted setting, the current implementations are not constant-time (see for instance [2]). The present research was motivated by this observation—although object-oriented technology is mature, scalable implementations are urgently needed due to the ever-increasing size of object-oriented class libraries and there is still considerable doubt about the scalability of existing implementations.

In a previous work, the first author proposed a new technique, called *perfect class hashing* (PH), for subtyping tests and method invocation [3]. To our knowledge, this is the first and only technique that fulfills all these requirements in the *multiple inheritance* and *dynamic loading* context. However, our experiments only concluded that the technique was promising and the need for further research was stressed. Two hashing functions were actually considered, namely *modulus*, i.e. the remainder of integer division, and bit-wise-`and`. These two functions involve a tradeoff between space and time efficiency, and the space efficiency of bit-wise-`and` needs to be further improved and assessed.

In this article, we present the results of more in-depth studies and experiments that show that a variant of perfect hashing, called *perfect class numbering*, provides a very efficient object-oriented implementation, even with bit-wise-`and` and from the space standpoint.

1.1. Object-oriented implementation and perfect hashing

One typical implementation issue of object-oriented languages is *late binding*, which is also referred to as the *message sending* metaphor. The underlying principle is that the address of the actually called procedure is not statically determined at compile-time, but depends on the dynamic type of a distinguished parameter known as the *receiver*. An issue similar to message sending arises with attributes (aka *fields*, *instance variables*, *slots*, *data members* according to the languages), since their position in the object layout may depend on the object's dynamic type. Subtyping introduces a last feature, i.e. run-time subtype checks, which amounts to testing whether the value of x is an instance of a class C or, equivalently, whether the dynamic type of x is a subtype of C. This is the basis for the so-called *downcast* operators. Object-oriented implementation represents the data structures and algorithms that are required for these three mechanisms, namely method invocation, attribute access and subtype testing.

When static typing is coupled with single inheritance of classes and types—we call it *single subtyping*—the standard implementation of object-oriented languages allows for direct access to the desired data at an invariant position. This optimizes the implementation. Otherwise, dynamic typing or multiple inheritance makes it harder to retain such direct access, especially in a dynamic loading setting.

Perfect Class Hashing[§] represents an alternative implementation in the context of static typing, multiple inheritance and dynamic loading. The position of all data required for implementing a class is no longer invariant with respect to the dynamic type of the object; it is instead determined by a collision-free hashing function that depends on the dynamic type. Such functions are called *perfect hashing* functions [4, 5].

Perfect class hashing was first applied to subtype testing. It can be formalized as follows. Let $(X, \preceq)$ be a partial order that represents a class hierarchy, namely X is a set of classes and $\preceq$

[‡]When the `virtual` keyword is omitted, the implementation is more efficient but no longer fully reusable because it yields *repeated inheritance*.The language is thus no longer compatible with both multiple inheritance and dynamic loading. In the following, we only consider C++ for the virtual semantics and implementation.

[§]The technique was originally called 'perfect hashing'. As this name generates confusion between the general hashing technique and its application to object-oriented languages, we now prefer to call it 'perfect class hashing'.

DOI: 10.1002/spe

the specialization relationship that supports inheritance. The subtype test amounts to checking at run-time that a class c is a superclass of a class d, i.e. $d \preceq c$. An efficient implementation of this test amounts to precomputing a data structure that allows for constant time. Dynamic loading adds a constraint, namely that the technique should preferably be incremental. Classes are loaded at run-time in a total order such that superclasses are loaded before subclasses. With an incremental technique, the data structure for the newly loaded class is computed from the data already computed for previously loaded classes, and no recomputation is needed. The *perfect class hashing* principle is as follows. When a class c is loaded, a unique immutable identifier id_c is associated with it and the set $I_c = \{id_d | c \preceq d\}$ of identifiers of all its superclasses is known. Thus, $c \preceq d$ iff $id_d \in I_c$. This set I_c is immutable; hence, it can be hashed with some *perfect hashing function* h_c, that is, a hashing function injective on I_c. The previous condition becomes: $c \preceq d$ iff $ht_c[h_c(id_d)] = id_d$, whereby ht_c denotes the hashtable of c, i.e. a simple array. The technique is obviously incremental since all hashtables are immutable and the computation of ht_c only depends upon I_c. In the same article, an application to method invocation in the restricted case of JAVA interfaces was also proposed.

1.2. Limitations of previous work

In our previous work, we considered one-parameter collision-free hashing functions such that $h_c(x) = hash(x, H_c)$, whereby H_c is the hashtable size. Two functions were considered for *hash*, namely *modulus* (denoted `mod` hereafter) and bit-wise-`and` (the exact function maps x to `and`$(x, H_c - 1)$). In both cases, H_c is defined as the least integer such that h_c is injective on the set I_c. The resulting techniques are, respectively, denoted as PH-`mod` and PH-`and`. Both functions involve a single machine instruction. However, whereas bit-wise-`and` is a 1-cycle instruction, the latency of integer division is commonly more than 20 cycles, hence markedly higher than the total cycle count of each mechanism with the standard single-subtyping implementation. Therefore, the time efficiency of PH-`mod` is expected to be rather low, whereas that of PH-`and` should be high. In the aforementioned article, we computed the H_c parameters on a set of large-scale benchmarks commonly used in the object-oriented compilation community. Our requirement for space-linearity means that the memory occupation for these tables should be linear in the cardinality of the specialization relationship $\preceq$. The results of our tests were encouraging but not perfect—namely, PH-`mod` required a little more than twice the cardinality of $\preceq$, but PH-`and` appeared to be much more greedy, especially in the single case of a very large benchmark. Hence, the scalability of PH-`and` was not certain, and the main point at issue is the memory occupation of PH-`and`.

Our assessment of time-efficiency relied on an abstract processor model borrowed from [6]; although we believe in the model validity, it should obviously be complemented by empirical time measurement. Later, we tested perfect hashing along with a variety of implementation techniques in the PRM testbed [7]. The results mostly confirm our earlier abstract evaluation.

These first experiments left, however, a number of issues open. PH parameters depend on the class IDs that are assigned to classes as they are loaded; hence, they should vary according to class-loading orders. However, only a single arbitrary order was considered, and it was likely a top–down depth-first linear extension that might be far from being representative of actual class loading orders. Efficient variants have also been sought. A variant based on *quasi-perfect hashing* (qPH) [8], which yields a 2-probe test, was proposed; it gives more compact tables at the expense of less efficient code. Another variant might be based on a 2-parameter hashing function, but we could not find a function that would reduce the table size while keeping the time overhead smaller than with modulus. We also proposed to optimize the identifier of the currently loaded class, but the technique, called *perfect class numbering* (PN), yielded strange results; hence, we did not include them in the article.

1.3. Contributions and plan

This article provides answers to these different issues:

- The testbed used for our previous experiments was extended in order to randomly generate class-loading orders. Potentially, any order can be generated, although the complete combinatorics is intractable. However, all class-loading orders are likely not sensible, and the statistics

DOI: 10.1002/spe

are also restricted to concrete class-loading orders, according to the assumption that only leaf classes can have instances—*'make all non-leaf classes abstract'* [9].
- *Perfect class numbering* is now tested in this new setting, which better explains the observed odd behavior of PN-`and`. Several 2-parameter functions have also been proposed and tested, but only a single one is retained.
- Finally, all these experiments are repeated with benchmarks of JAVA hierarchies that include precise information about interfaces and abstract, concrete and inner classes.

Furthermore, we present some mathematical properties of PH and PN that are the basis of simple and efficient algorithms, and also prove that PN is optimal in single-inheritance situations.

Overall, these new tests overcome all our previous reservations about the use of perfect hashing for implementing object-oriented languages. In a dynamic loading setting, we now consider that the technique is efficient, under its PN-`and` form, from both space and time standpoints. In contrast, we also think that modulus is too inefficient in terms of time to be further considered. Therefore, although we developed the same formal and empirical framework for modulus, here we only present the bit-wise-`and` results. Thus, perfect hashing should be considered by language implementers (i) for subtype testing in all languages with multiple inheritance, (ii) for implementing interfaces in all languages with multiple subtyping (e.g. JAVA, C# , ADA 2005, etc.). However, further research and tests are required before using perfect hashing for attribute access, hence complete object implementation.

The article is structured as follows. The next section presents the question of object-oriented implementation in a static typing setting. The standard single-subtyping implementation is described and several possible extensions to multiple inheritance are discussed: (i) the subobject-based implementation used by C++; (ii) the coloring approach that amounts to maintaining single-subtyping invariants in a global setting and finally (iii) the perfect class hashing approach. Section 3 presents definitions and simple mathematical results that provide efficient algorithms and lower bounds that are proven to be the PN value in case of single inheritance. Section 4 presents experiments and statistical results in the context of full multiple inheritance, and then in the context of JAVA interfaces. All these experiments concern space-efficiency. Section 5 briefly describes experiments conducted in the PRM compiler and summarizes this first empirical assessment of the PH run-time time-efficiency. Related works and current interface implementations are discussed in Section 6. Finally, Section 7 presents conclusions and prospects. An appendix describes algorithms for computing PH parameters.

This article is an continuation of [3]. We tried to make it as self-contained and short as possible, but we sometimes refer to the original article—hereafter it will be abbreviated PHAPST (Perfect Hashing as an Almost Perfect Subtype Test). Interested readers are referred to PHAPST for a more in-depth presentation and discussion of subtype testing and all related topics, and to [10] for a review of object-oriented implementation.

2. FROM SINGLE SUBTYPING TO MULTIPLE INHERITANCE

Single subtyping, i.e. single inheritance and static typing, allows for a straightforward constant-time implementation of the basic object-oriented mechanisms. We first describe this implementation, then explain why it does not easily generalize to multiple inheritance or dynamic typing. Finally, we present some alternatives, namely C++ subobjects, coloring and perfect hashing.

2.1. Single subtyping

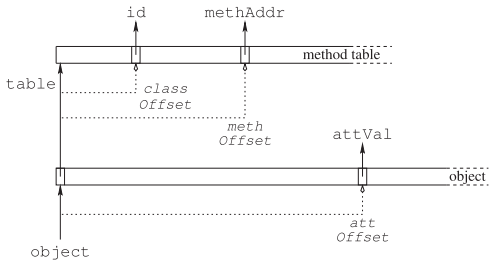
2.1.1. Method invocation and object layout. In separate compilation of statically typed languages, late binding is generally implemented with tables that are called *virtual function tables* in C++ jargon. An object is laid out as an attribute table, with a header pointing at the class table that contains method addresses. Method calls are then reduced to calls to pointers to function through two extra indirections. With single inheritance, the class hierarchy is a tree and the tables implementing a class are straightforward extensions of those of its single superclass. Each class

DOI: 10.1002/spe

```
// attribute access
load [object + #attOffset], attVal

// method invocation
load [object + #tableOffset], table
load [table + #methOffset], methAddr
call methAddr

// subtype test
load [object + #tableOffset], table
load [table + #classOffset], id
comp id, #targetId
bne #fail
// succeed
```



Code sequences for the 3 basic mechanisms and the corresponding diagram of object layout and method table. Pointers and pointed values are in Roman type with solid lines, and offsets are italicized with dotted lines.

Figure 1. Single-subtyping implementation: data structure and code sequences.

adds, to (a copy of) the tables of its direct superclass, the entries required for implementing the properties, i.e. attributes or methods, that are *introduced* by the considered class (a class A *introduces* a property if A defines it as a new signature that does not correspond to any property of its superclasses). Figure 1 presents a diagram of the object layout and method table in this setting, together with the corresponding code sequences in an intuitive pseudo-code borrowed from [6].

This 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 tables does not depend on the dynamic type of the object. Therefore all accesses to objects are straightforward, but this simplicity requires both static typing and single inheritance. From a spatial standpoint, the object layout is optimal, since there is one field per attribute, with a single extra pointer to the method table, which shares the data common to all direct instances of the considered class. The method tables are not very small, but they are also somewhat optimal. Based on the assumption that the method introduction is uniformly distributed over all classes, the total size of method tables is linear in the size of the class specialization relationship, which is assumed to be reflexive and transitive. In the worst case, this is however quadratic in the number of classes.

2.1.2. Subtype tests. The subtype test amounts to checking at run-time that a class c is a superclass of a class d, i.e. $d \preceq c$. Usually d is the dynamic type of an object, and the programmer or compiler wants to check that this object is actually an instance of c. In the single-inheritance context, several techniques have been proposed and are commonly used. We present only one of the simplest ones that provides a basis for further generalizations. The technique is known as *Cohen's display* and was first described in [11]. It consists of assigning two integers to each class, a unique ID and a non-unique offset. Let the ID of class C be id_C and the offset of C be $\delta(C)$. Each subclass of a class C has id_C stored in its method table at $\delta(C)$. Thus, an object is a direct or indirect instance of a class C if and only if offset $\delta(C)$ in its method table (*tab*) contains id_C. This is equivalent to testing that the class, say D, that has instantiated the considered object is a subtype of C:

$$D \preceq C \Leftrightarrow tab_D[\delta(C)] = id_C \tag{1}$$

Originally, Cohen's display required a set of tables separate from the method tables, and $\delta(C)$ was the depth of C in the class hierarchy. However, in a statically typed language, these tables can be merged within method tables. Class offsets are ruled by the same position invariant as methods and attributes. This can be thought of as giving each class C a method for checking whether an object is an instance of C, i.e. such that its instances can check that they are. The test fails when this pseudo-method is not found. When inlining Cohen's display in the method table, some precautions are required to avoid bound checks and possible confusion between class IDs and method addresses without consuming unnecessary memory. This is discussed in PHAPST.

DOI: 10.1002/spe

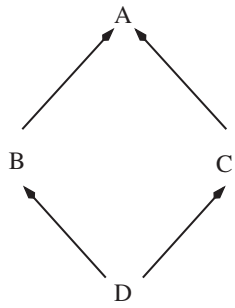


Figure 2. The diamond example of multiple inheritance.

2.2. Multiple inheritance

With multiple inheritance, both invariants of reference and position cannot hold together, at least if compilation—i.e. computation of positions—is to be kept separate. For instance, in the *ABCD* diamond hierarchy of Figure 2, if the implementations of *B* and *C* simply extend that of *A*, as in single subtyping, the same offsets might be occupied by different properties in *B* and *C*, thus prohibiting a sound implementation of *D*. Therefore, invariants or separate computation must be dropped.

2.2.1. Multiple inheritance with static typing and dynamic linking. The 'standard' implementation of multiple inheritance in a static typing and separate compilation setting (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. Each subobject contains attributes *introduced* by the corresponding class, together with a pointer to a method table that contains methods *known* by the class. Both invariants are dropped, as both reference and position now depend on the current static type. This is the C++ implementation when the `virtual` keyword annotates each superclass. It is time-constant and compatible with dynamic loading, but method tables are no longer space-linear. Indeed, the number of method tables is exactly the size of the specialization relationship. Hence, the worst-case total table size of a class is now quadratic in the number of superclasses, and the total size for all classes is cubic in the class number. Furthermore, all polymorphic object manipulations—i.e. assignments and parameter passing, when the source type is a subtype of the target type— require *pointer adjustments* between source and target types, as they correspond to different subobjects. These adjustments can be done with offsets in method tables or explicit pointers in the object layout.

A detailed presentation of this implementation is beyond the scope of this article. Interested readers are referred to [1] for C++ itself, [10] for a language-independent analysis of the implementation and [12] for an analysis of multiple inheritance. Anyway, there are three main drawbacks: (i) pointer adjustments represent ubiquitous overhead throughout programs; (ii) subobjects add many compiler-generated fields in the object layout and (iii) the total size of method tables is now, in the worst case, cubic in the number of classes, instead of quadratic. Overall, this seems to be the price to be paid for multiple inheritance in a static typing and separate compilation setting. Finally, the main drawback of this implementation family is that its overhead remains even when multiple inheritance is not used. Therefore, language designers have provided alternative specification and implementation, known as *non-virtual inheritance*, when omitting the `virtual` keyword. Non-virtual inheritance gives exactly the same implementation as single subtyping in the case of single-inheritance hierarchies, but it provides degraded semantics of general multiple inheritance, hence preventing sound reusability—for instance, in the *ABCD* diamond example, the attributes introduced by *A* would be duplicated in the object layout of *D*.

2.2.2. Coloring for multiple inheritance with global linking. The general idea of coloring is to keep the two *invariants* of single inheritance, i.e. *reference* and *position*. An injective numbering of attributes, methods and classes verifies the position invariant; hence, this is clearly a matter of

DOI: 10.1002/spe

optimization for minimizing the size of all tables—or, equivalently, the number of *holes*, i.e. empty entries. However, this optimization must be done globally, at compile- or link-time. Thus, the technique is not incremental and not directly compatible with dynamic loading. Coloring has been independently proposed for the three mechanisms that underlie object-oriented implementation. *Method coloring* was first proposed by Dixon *et al.* [13] under the name of *selector coloring*. Coloring was applied to attributes by Pugh and Weddell [14] and to classes by Vitek *et al.* [15] (under the name of *pack encoding*). Interested readers are referred to [16], which reviews the approach.

2.2.3. Perfect class hashing for object implementation. So far, the current situation of object-oriented implementation is as follows: (i) single-subtyping implementation meets all requirements, is compatible with dynamic loading but not with multiple inheritance; (ii) coloring, which represents its extension to multiple inheritance, is no longer compatible with dynamic loading; (iii) subobject-based implementation is compatible with both multiple inheritance and dynamic loading; however, its space requirements are cubic in the number of classes and ubiquitous pointer adjustments imply marked run-time overhead.

Perfect class hashing represents an alternative approach, first applied to subtype testing. Its principle is as follows. Classes are loaded at run-time in some total order that must be a *linear extension* (aka *topological sorting*) of $(X, \preceq)$—that is, when $d \prec c$, c must be loaded before d. As an alternative view, when a class is loaded, its yet unloaded superclasses are recursively loaded. Anyway, when a class c is loaded, a unique and immutable identifier id_c is associated with it and the set $I_c = \{id_d | c \preceq d\}$ of the identifiers of all its superclasses is known. Thus, $c \preceq d$ iff $id_d \in I_c$. It is worth noting that the class ID is some arbitrary integer. Any injective class numbering can work, but the overall implementation may constrain class IDs to be a specific integer subset. Actually, as we shall see, the most promising scheme involves computing the ID of a class according to the IDs of its superclasses. For the moment, let us consider a simple numbering, starting from 0.

The set I_c is immutable and can be hashed with some *perfect hashing function* h_c, i.e. a hashing function that is injective on I_c [4, 5]. The previous condition becomes: $c \preceq d$ iff $ht_c[h_c(id_d)] = id_d$, whereby ht_c denotes the hashtable of c. The hashing function family h_c is parameterized by the hashtable size H_c, such that $h_c(x) = hash(x, H_c)$, where *hash* is some low-level operation. In PHAPST, we considered modulus (`mod`) and bit-wise-`and`. Both involve a single machine instruction. The technique is obviously incremental since all hashtables are immutable and the computation of ht_c only depends upon I_c. It is also time-constant and inlinable. The only point might be space-linearity, but the conclusions of PHAPST were rather positive if not definitive.

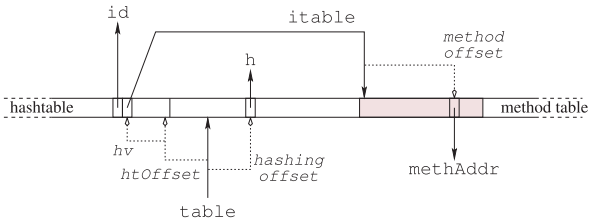
In a static typing setting, the technique can also be applied to method invocation and we proposed, in the aforementioned article, an application to JAVA interfaces. For this, as hashing method identifiers appeared to be over space-consuming, the hashtable associates, with each implemented interface, the address (or offset) of the group of methods that are introduced by the interface. Figure 3 recalls the precise implementation in this context and the corresponding code sequence. The method table is bidirectional. Positive offsets involve the method table itself, organized as with single subtyping. Negative offsets consist of the hashtable, which contains, for each implemented interface, a two-fold entry. The object header points at its method table via the `table` pointer. `#hashingOffset` is the position of the hash parameter (`h`) and `#htOffset` is the beginning of the hashtable. At a position `hv=and(h,#hashingOffset)` in the hashtable, a two-fold entry is depicted that contains both the implemented interface ID that must be compared with the target interface ID (`#interfaceId`), and the address `itable` of the group of methods introduced by the interface that introduces the considered method. The table contains the address of the function that must be invoked, at the position `#methodOffset` determined by the considered method in the method group. This method invocation technique requires static typing. However, it does not require a method to be introduced by a single class. If a method is introduced by several classes (or interfaces, as is possible in JAVA), the method entry will be replicated in each method group, with the same method address.

DOI: 10.1002/spe

```
//preamble for both mechanisms
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #interfaceId, h, hv
sub table, hv, htable

//subtyping test
load [htable+#htOffset-fieldLen], id
comp #interfaceId, id
bne #fail

//method invocation
load [htable +#htOffset], itable
load [itable +#methOffset], method
call method
```



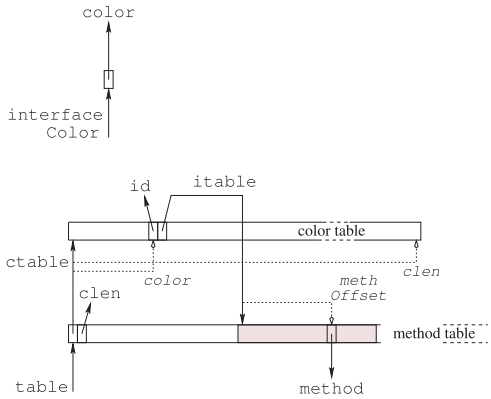
`fieldLen` represents the integer size, e.g. 4 if 32-bit integers are used. In practice, all numbers (i.e. H and class ID's) must be multiplied by `2*fieldLen`. Of course, it works (i.e. it commutes with `and`) because it is a power of 2. The grey rectangle denotes the group of methods introduced by the considered interface.

Figure 3. PH-`and` for JAVA interfaces, code sequence and object representation.

```
// preamble
load [object + #tableOffset], table
load [table + #ctableOffset], ctable
load [#interfaceColor], color
add ctable, color, ctable

// subtyping test
load [table + #ctableOffset+4], clen
comp color, clen
bgt #fail
load [ctable], id
comp #interfaceId, id
bne #fail

// method invocation
load [ctable +#fieldLen], itable
load [itable +#methOffset], method
call method
```



When coloring is incremental, the method table itself (pointed by `table`) is constant but color tables (`ctable`) and colors (`color`) may be recomputed at load-time. Hence three disconnected memory areas are involved and a bound check is required for subtype testing (see PHAPST for an in-depth discussion of this implementation).

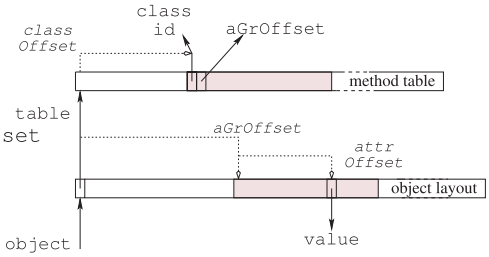
Figure 4. Incremental coloring of JAVA interfaces.

With rooted hierarchies, a slight optimization is possible. The position of the group of methods introduced by the root can be made invariant, and invoking such a method only requires the SST code sequence. The root can even be removed from the hashtable. Indeed, when the target is statically known, subtype testing is useless if it is the root. In contrast, when the target is not statically known (e.g. for implementing covariance), the test will only require an extra equality test against the root ID.

2.2.4. Incremental coloring (*IC*). Although coloring is not inherently incremental, its use for subtype testing has been proposed in the context of JAVA interfaces [17]. We called it *incremental coloring* (IC) in PHAPST, and extended its use to method invocation in the same way as for perfect hashing. This use of coloring with both dynamic loading and multiple inheritance yields marked overhead at load-time, since some recomputation may be required, and at run-time, since these possible recomputations increase the number of memory accesses, while degrading their locality (Figure 4). Overall, our abstract estimation was that IC should not be better than perfect hashing, or at least not sufficiently better to offset the expected load-time overhead.

DOI: 10.1002/spe

```
// attribute access
load [object + #tableOffset], table
load [table + #classOffset+fieldLen], aGrOffset
add object, aGrOffset, attgr,
load [attgr + #attrOffset], value
```



The object representation with accessor simulation coupled with class and method coloring must be compared with Fig. 1. The offset of the group of attributes introduced by a class (`aGrOffset`) is associated with its class ID in the method table, and an attribute position is now determined by an offset (`#attrOffset`) relative to this attribute group.

Figure 5. Accessor simulation with method coloring.

2.2.5. Accessor simulation (*AS*). Some techniques, such as perfect hashing and IC, apply only to method invocation and subtype testing. Accessor simulation (AS) is a way of applying them to attribute access. An accessor is a method that either reads or writes an attribute. Suppose that all accesses to an attribute are through an accessor. Then the attribute layout of a class does not have to be the same as the attribute layout of its superclass. A class will redefine accessors for an attribute if the attribute has a different offset in the class than it does in the superclass. True accessors require a method call for each access, which can be inefficient. However, a class can simulate accessors by replacing the method address in the method table with the attribute offset. This approach is called *field dispatching* by Zibin and Gil [18]. Another improvement is to group attributes together in the method table when they are introduced by the same class. Then one can substitute, for their different offsets, the single relative position of the attribute group, stored in the method table at an invariant position, i.e. at the class color with coloring (Figure 5) [10, 19]. With PH and IC, the attribute–group offset is associated with the class ID and method–group address in the hashtable or color-table, yielding 3-fold table entries.

AS is a generic approach to attribute access that works with any method invocation technique; only grouping can be conditioned by static typing, since attributes must be partitioned by the classes that introduce them.

3. PERFECT CLASS HASHING AND NUMBERING

In this section, we formally define perfect class hashing (PH) and numbering (PN) and present simple theoretical results that are the basis for efficient algorithms.

3.1. Definitions

Definition 3.8 (*Perfect hashing*)
Let I be a non-empty set of integers, and $hash: \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ be a function such that $hash(x, y) < y$ and $hash(x, y) \leq x$ for all $x, y \in \mathbf{N}$. Then, the *perfect hashing parameter* of I is the least $H \in \mathbf{N}$ such that the function h that maps x to $h(x) = hash(x, H)$ is *injective* on I, i.e. for all $x, y \in I$, $h(x) = h(y)$ implies $x = y$.

The definition is extended to empty sets by considering that $H = 1$ when $I = \emptyset$.

This least integer exists for `mod` and `and` (the exact function maps x to `and`$(x, H-1)$), but likely not for any *hash* function. Moreover, $H \geq n$, where n is the cardinality of I.

Perfect class hashing (PH). Let $(X, \preceq)$ be a class hierarchy equipped with some injective class identifier $id: X \to \mathbf{N}$. Then, perfect hashing applies to each class c in X by considering the set $I_c = \{id_d | c \preceq d\}$. The resulting parameter H_c is the size of the hashtable (a simple array) that implements class c, and for each superclass d, this table contains id_d at position $h_c(id_d)$. All other

DOI: 10.1002/spe

positions j contain any integer l such that $h_c(l) \neq j$. The $hash(x, y) \leq x$ constraint is not strictly necessary, but it is verified by all the tested functions. It also implies that $h_c(0) = 0$ for all c. With rooted class hierarchies, 0 is thus a convenient identifier for the root, and it is also a convenient value for empty entries at offset $j > 0$. Conversely, with unrooted hierarchies, any non-null integer can represent an empty entry at offset 0, but this integer must not be used as a class ID.

In the following, the notations I, H, h and n will be implicitly used in reference to Definition 3.8, with a subscript indicating that set I actually consists of class identifiers.

Proposition 3.2 (*Monotonicity*)
Perfect hashing is monotonic, that is, if $I \subset I'$, then $H \leq H'$.

The perfect class hashing definition holds for any injective class identifier, for instance a consecutive numbering as classes are loaded. An optimization of perfect class hashing amounts to optimizing the class identifier id_c in order to minimize the H_c parameter. Thus, instead of numbering classes as they are loaded, by incrementing a counter, the H_c parameter is first computed from the superclass identifiers, before computing the best id_c that fits the resulting hashtable.

Definition 3.3 (*Perfect numbering*)
Let I' be a set of integers, and F be a set of *free* integers, disjoint from I'. Then the perfect numbering parameter is defined as the least $H \in \mathbf{N}$ such that: (i) there is a free integer id in F, and (ii) H is the perfect hashing parameter for the set $I = I' \uplus \{id\}$.

Perfect class numbering (PN). Then, perfect numbering applies to each class c in X by considering the set $I'_c = \{id_d | c \prec d\}$, and F is the set of integers that are not identifiers of previously loaded classes. In practice, F is large enough to ensure that H does not depend on F. The resulting parameter H_c is the size of the hashtable that implements class c and id_c can be defined as the least number in F such that H_c is the perfect hashing parameter for $I_c = I'_c \uplus \{id_c\}$. The code generated for PN is exactly the same as that for PH (Figure 3); hence, they have the same time-efficiency. The only difference involves computation of the H_c and id_c parameters, and this computation involves only a slight overhead over that of PH (see Appendix A.2.). Hence, PN must be preferred to PH if the expected space improvement is effective.

As loading a class triggers the loading of yet unloaded superclasses, it is interesting to extend this definition to the allocation of several identifiers.

Definition 3.4 (*Perfect k-numbering*)
Let $k > 0$ be an integer, I' be a set of integers and F be a set of *free* integers, disjoint from I'. Then the perfect k-numbering parameter is defined as the least $H \in \mathbf{N}$ such that: (i) there is a set of k free identifiers $I'' \subset F$, and (ii) H is the perfect hashing parameter for the set $I = I' \uplus I''$.

Perfect class k-numbering. Now, perfect numbering applies to each class c in X as follows. Let $X' \subset X$ be the current subset of already loaded classes, $id : X' \to \mathbf{N}$ an injective class identifier function, and $F = \mathbf{N} \backslash id(X')$ the set of free identifiers. Let $c \in X \backslash X'$ be some unloaded class, with $X'_c = \{y \in X' | c \prec y\}$ and $X''_c = \{y \in X \backslash X' | c \preceq y\}$. Then, $I'_c = id(X'_c)$ and $k \geq 1$ is the cardinality of X''_c. H_c and I''_c are successively computed, and the numbers in I''_c are finally assigned to classes in X''_c.

The resulting parameter H_c is the size of the hashtable that implements c, and it is uniquely determined. There are, however, many ways to define I''_c as a 'minimal' set and to map I''_c to newly loaded superclasses. Moreover, only H_c is optimized, not other H_x parameters when $c \prec x$ and x is not already loaded. It might be desired if c is a concrete class, while x is abstract and does not require a method table. However, an optimization of H_x might improve the space occupation of other concrete subclasses of x. Therefore, the optimization of each H_x would be better. The previous notations are extended as follows: for each $x \in X''_c$, X'_x and X''_x represent, respectively, the sets of previously loaded and yet unloaded superclasses of x, and H_x is the perfect $|X''_x|$-numbering parameter of $id(X'_x)$.

DOI: 10.1002/spe

Conjecture 3.5 (*Optimal perfect class k-numbering*)
There is an injective mapping $f : X''_c \to F$ such that the PH parameter of $id(X'_x) \cup f(X''_x)$ is H_x for each $x \in X''_c$.

3.2. Application to bit-wise-and

We now focus on the case where $hash(x, y) = \texttt{and}(x, y-1)$. The effects of bit-wise hashing are not always intuitive, and we present some very simple results that might help readers. Furthermore, perfect class hashing involves a simple lower bound along with a fine optimality condition. Let I be a set of n positive or null integers, m be an integer that serves as a bit-wise mask in the function $h(x) = \texttt{and}(x, m)$ and $H = m+1$ the perfect hashing parameter, i.e. the hashtable size.

Definition 3.6 (*Minimal mask*)
Given a set I of integers, a mask m is minimal for I if m provides an h function injective on I and if switching any 1-bit in m makes h non-injective.

The mask corresponding to Definition 3.8 is minimal, but all minimal masks do not yield perfect hashing in the sense of this definition.

The next two propositions give lower and upper bounds to the number of 1-bits in the mask, as a function of n. Of course, they yield a lower bound but no upper bound to the mask itself.

Proposition 3.7 (*PH-and lower bound*)
Hashing $n > 0$ integers requires a mask with at least $\log_2 n$ 1-bits; hence, $2^{\lceil \log_2 n \rceil} \in [n, 2n[$ is a lower bound for the H parameter.

The proof follows from the fact that a mask with k 1-bits can discriminate exactly 2^k integers.

The point is that the hashtable capacity depends on the 1-bit count of the mask, not on the 1-bit positions, which determine its magnitude, that is, the hashtable size. Therefore, with PH-and, a hashtable may have a lot of empty entries, while being full in the sense that no other number can be hashed within it. In contrast, with mod, a hashtable of size H could hash H integers. In other words, with and and unlike mod, h is not surjective in the $[0, H-1]$ range. This certainly accounts for the different behaviors of both hashing functions and, in practice, it prevents us from using the same algorithms for both.

Proposition 3.8 (*PH-and 1-bit count upper bound*)
A mask m minimal for a set I with cardinality n has at most $n-1$ 1-bits, and this upper bound is reachable.

The proof is by induction. It is trivial for $n = 1$. Suppose now that it is true for any set of cardinality $n-1$. Let I be a set of n integers and x be the maximum element of I. Suppose that a mask m with n 1-bits is minimal for hashing I. Consider now the set $I' = I \backslash \{x\}$. According to the recurrence assumption, m is not minimal for I' and two 1-bits in m can be switched—let m' be the resulting minimal mask for I'. As m' does not make a perfect hashing function on I (otherwise we get the proof), there is a single $y \in I'$ that agrees with x on all 1-bits of m'. Hence, x and y must differ on the two bits that are 1 in m and 0 in m' but only one is required to make a perfect hashing mask since y is unique. Hence, m is not minimal. The upper bound is reachable when each number differs from all others by exactly one 1-bit—for instance, if $I = \{2^i | i = 0..n-1\}$.

The following proposition gives an upper bound as a function of the maximum element of I.

Proposition 3.9 (*PH-and magnitude upper bound*)
A minimal mask m is strictly less than $2^{\lceil \log_2(\max(I)+1) \rceil} \leq 2\max(I)$ and the $(2\max(I)-1)$ bound is reachable.

The minimal mask is bounded by the integer formed by all the 1-bits of all integers in I. Thus, an upper bound is formed by a chain of k consecutive 1-bits, where $k-1$ is the rank of the leftmost

DOI: 10.1002/spe

1-bit in $\max(I)$. In the worst-case, all these 1-bits are required to form the mask—for instance, if $k=n-1$ and $I=\{0\}\cup\{2^i|i=0..n-2\}$.

Proposition 3.10 (*PH-*`and`* global upper bound*)
Let X be a class hierarchy with N classes and an injective identifier $id: X\to[0, N-1]$, then $\sum_{i=1..N} 2^{\lceil\log_2(i+1)\rceil}$ is an upper bound of $\sum_c H_c$.

This is a corollary of Proposition 3.9. However, as $2^{\lceil\log_2(i+1)\rceil}$ is strictly greater than i, this upper bound is in the range $[N(N+1)/2, N(N+1)]$; hence, it is too high to be useful.

Given a mask m, the $m+1$ hashtable entries can be partitioned into three sets: the entries occupied by the input numbers I; unusable entries that do not fit with the mask m and can be allocated for any data; free entries that could be occupied by new numbers. The next two propositions explain it.

Proposition 3.11 (*PH-*`and`* unusable entries*)
Let m be a mask and k be the number of 1-bits of m. Then the corresponding hashtable contains $m+1-2^k$ entries in which no number can be hashed.

Let m' be the mask formed by complementing the significant bits of m, i.e. 1-bits of m are 0 in m', and 0-bits of m are 1 in m' if they are on the right of some 1-bit. Then, for any $j\in[1..m-1]$ such that $\texttt{and}(m', j)\neq 0$, $h^{-1}(j)=\emptyset$.

Proposition 3.12 (*PH-*`and`* completion*)
Let I be a set of n integers and m be a mask that forms a perfect hashing function on I. Let k be the number of 1-bits of m. When $n<2^k$, there is an infinity of sets J of cardinality 2^k-n, disjoint from I, such that m forms a perfect hashing function on $I\uplus J$.

This is a direct consequence of Proposition 3.7. Among the 2^k combinations of 1-bits of m, only n are used by I and the remaining is free and can be used for integers in J.

This means that any set can be completed to reach a 2^k cardinality while keeping the same optimal bit-wise mask. The perfect numbering algorithm follows from the next proposition.

Proposition 3.13 (*Unbounded PN-*`and`)
Let I' be a set of integers of cardinality n', and $F=\mathbf{N}\backslash I'$ be the set of *free* integers. Let m' be the PH mask of the set I'. Then the PN mask m is m' if the bit-wise mask m' has at least $\log_2(n'+1)$ 1-bits. Otherwise, $n'=2^k$ and m is m' with its least-weight 0-bit switched to 1. As a corollary, $H\leq 2H'$, whereby H' is the perfect hashing parameter of the set I'.

The first part is a straightforward consequence of Proposition 3.12. The corollary is implied by the fact that, in the worst-case, the least-weight 0-bit is just after the leftmost 1-bit.

This ensures that there is a subset of 1-bits in m that forms the offset of an empty entry in the new table. This proposition holds if the set F of free numbers is large enough, for instance for any finite class hierarchy with an initial set of free integers that consists of all 16- or 32-bit integers. It is worth noting that both `and` and `mod` functions present the same $H\leq 2H'$ upper bound.

Moreover, PN-`and` has an interesting property—namely it is optimal, i.e. $H_c=2^{\lceil\log_2(n_c)\rceil}$, for all classes c in single inheritance, that is, such that c and all its superclasses have a single direct superclass. This is a limited converse of Proposition 3.7.

Proposition 3.14 (*PN-*`and`* optimum*)
Perfect numbering is optimal, i.e. $H_c=2^{\lceil\log_2(n_c)\rceil}$, for every class c in single inheritance, i.e. such that c and all its superclasses have a single direct superclass.

This means that, in single inheritance, all masks are formed by a chain of $2^{\lceil\log_2(n_c)\rceil}$ rightmost consecutive 1-bits. The proof is by induction on n_c, i.e. class depth. Let c be the considered class, and c' be its single direct superclass, with $n_{c'}=n_c-1$. By induction, $H_{c'}=2^{\lceil log_2(n_c-1)\rceil}$. If $n_c=2^k+1$, $H_{c'}=2^k$ and the mask is full, then an extra bit is required and $H_c=2^{k+1}$. Otherwise,

DOI: 10.1002/spe

according to Proposition 3.12, the mask has some free room for an extra identifier and it can be inherited by c; hence, $H_c = H_{c'} = 2^{\lceil \log_2(n_c) \rceil}$.

This proposition holds under the same condition as Proposition 3.13 on the set F of free integers. The interesting point is that this perfect class 1-numbering optimum does not depend on the exact class identifiers, as any greedy choice provides a global optimum. Intuitively, all 1-bits of the mask form a rightmost prefix of the mask and are 'inherited'. With multiple inheritance, when a class c has two direct superclasses c_1 and c_2, each one with a mask formed by a chain of, respectively, k_1 and k_2 1-bits with $k_1 \leq k_2$, the k_2 1-bits of the highest mask cannot always discriminate between the extra identifiers inherited from c_1, either because there are not enough 1-bits, or because there are some multiple inheritance conflicts, that is, the same 1-bits are used by each class for discriminating its proper ancestors. Hence, extra bits are required, which can be much more leftward and yield an exponential increase in the mask. This explains why PN and PH can be erratic with bit-wise-`and` in multiple inheritance. Proposition 3.14 is important because all multiple inheritance class hierarchies are mostly in single inheritance—see, for instance, the statistics in [16]. Hence, PN-`and` should be optimal on a large part of each hierarchy and this should counterbalance its worst-case behavior on the multiple inheritance core.

Finally, the optimality of PN-`and` with single inheritance yields formal space-linearity.

Proposition 3.15 (*Space linearity*)
For any single-inheritance hierarchy, the PN-`and` ratio $\rho = \sum_c H_c / \sum_c n_c$ is in interval $[1, 2[$. The upper bound is asymptotically reachable, that is, for all $\varepsilon > 0$, there are single-inheritance class hierarchies such that $\rho > 2 - \varepsilon$.

Consider, for instance, a chain $A_{2^k} \prec A_{2^k-1} \prec \cdots \prec A_1$ of 2^k classes, with A_{2^k} having x direct subclasses. For any such subclass c, $n_c = 2^k + 1$, $\lceil \log_2 n_c \rceil = k + 1$ and $H_c = 2^{k+1}$. Hence, in this framework, x can be large enough to make $\rho > 2^{k+1}/(2^k + 1)$.

In practice, the optimal ratio $\sum_c 2^{\lceil \log_2(n_c) \rceil} / \sum_c n_c$ is always lower than 1.5 in our multiple inheritance benchmarks, and $\sum_c H_c / \sum_c n_c$ can be very close to it in the best cases, but markedly higher in the worst cases (see Table II).

Discussion on perfect numbering. Its mathematical definition (Definition 3.8) corresponds to a simple algorithm (Proposition 3.13) that does not depend on the set F of free integers. In practice, class identifiers are bounded by the underlying integer implementation, e.g. 16- or 32-bit integers, and it might happen that both mathematical and algorithmic definitions differ when no free integers fit the hashtable in the algorithm. However, such a situation would be very unlikely, as the greatest allocated ID is never greater than 3 times the class number N in our tests.

Whereas the perfect class hashing definition (Definition 3.8) results in a $\sum_c H_c$ value that is uniquely determined by the input class numbering, perfect numbering does not provide the same overall determinism, apart from the single-inheritance optimum. Indeed, the perfect numbering parameter provides only a step-by-step optimal H_c, and there are several possible *ids* for the newly loaded class c. Selecting the least *id* amounts to a greedy optimization that does not ensure any formal minimization of $\sum_c H_c$. The point is even more complicated with k-numbering, since Proposition 3.13 no longer provides a step-by-step optimum. Conjecture 3.5 is quite attractive, since it relies on computation of the PH optimum for each currently loaded class.

Proposition 3.8 (*Optimal perfect class k-numbering*)
Conjecture 3.5 does not hold for bit-wise-`and`.

Although it may work for some class-loading orders of very large hierarchies (e.g. the Eclipse hierarchy in Table VII), we found simple counter-examples of the existence of a solution.

In the general case, the perfect class k-numbering problem is over-constrained and has no optimal solution, i.e. regardless of the classes that might be loaded afterwards. Only greedy optimizations can thus be considered. Moreover, while PN is markedly better than PH on average, we also

DOI: 10.1002/spe

observed some class-loading orders where PH was better than PN. Hence, the heuristics might well be improved by further research.

3.3. Other hashing functions

Without loss of generality, the perfect hashing approach might be improved with small variations such as: (i) 2-probe tests, (ii) other 1-parameter 1-operation *hash* functions or (iii) 2-parameter 2-operation functions. In PHAPST, we tested 2-probe quasi-perfect hashing (qPH-`and`); the space improvement was not marked enough to offset the expected time overhead. We could not imagine other 1-parameter operations than `mod` and `and`. Two-parameter *hash* functions might be an improvement from the space standpoint, but the extra instruction would likely degrade the time efficiency even if it takes a single cycle. Overall, the only efficient combination that we could imagine is bit-wise-`and` with rightward `shift`, in order to truncate the mask and remove all trailing zeros. This is denoted by PH-`and+shift`.

Definition 3.17 (*PH-`and+shift`*)
Let I be a non-empty set of integers. Then, the PH-`and+shift` *parameters* are defined as the pair (H_1, H_2) such that H_1 is the least integer such that there is H_2 such that the function h that maps x to $h(x) = \texttt{and}(\texttt{shift}(x, H_2), H_1 - 1)$ is *injective* on I.

An algorithm is proposed in Appendix A.3. Moreover, `and` and `shift` can also be combined with perfect numbering. The technique, denoted as PN-`and+shift` , amounts to substituting PH-`and+shift` for PH-`and` in the algorithm for PN-`and`, with the only difference that the switched 0-bit is selected between existing 1-bits, if possible, instead of being the least-weight 0-bit.

4. SPACE-EFFICIENCY TESTS

Our original testbed consists of a set of large-scale benchmarks that are commonly used in the object-oriented compilation community, together with a set of programs for computing various parameters that are either characteristics of the class hierarchies or the size requirement of various implementation techniques. This testbed thus simulates the memory occupation of these various techniques. The simulation is exact and reproducible, except for techniques that rely on heuristics like coloring or depend on some hidden run-time inputs, for instance class-loading orders. Besides PHAPST, this testbed has been used in different simulations [10, 16].

For this article, we have extended this testbed in several directions, by computing random class loading orders; by testing the new perfect hashing variants described in Section 3; by restricting class-loading orders to leaf or concrete classes; finally, by extracting a set of new JAVA benchmarks that retain all class modifiers, unlike the benchmarks that are commonly used in the literature.

4.1. Class loading at random

First of all, we computed various perfect hashing parameters while loading classes at random, with the same benchmarks as in PHAPST. They represent full-multiple-inheritance hierarchies that have been extracted from real libraries of different languages. Only the class specialization relationship is considered here. With JAVA hierarchies, classes and interfaces are not distinguished from each other. Table I presents, for each hierarchy, its class number N, then the average value of n_c, the number of superclasses of c, per class (i.e. $|\preceq|/N$).

The precise testbed involves generating a random class-loading order by selecting a class at random from the set of $\preceq$-*maximal* yet unloaded classes until all classes are loaded. Then, for each class-loading order, all PH parameters are computed. This is repeated thousands of times.

The statistics presented in Tables II and IV present minimum, average and maximum values over randomly generated class-loading orders. All these numbers are normalized ratios $\rho = \sum_c H_c / \sum_c n_c$ that represent the space-linearity factor, since $\sum_c n_c = |\preceq|$. The order of magnitude of ρ can be analyzed in several ways. Of course $\rho \geq 1$ for all hashing functions, and the

DOI: 10.1002/spe

Table I. Statistics on class and superclass numbers.

	Class number	Avg. n_c
IBM-SF	8793	9.2
JDK1.3.1	7401	4.4
Java.1.6	5933	4.3
Orbix	2716	2.8
Corba	1699	3.9
Orbacus	1379	4.5
HotJava	736	5.1
JDK.1.0.2	604	4.6
Self	1802	30.9
Geode	1318	14.0
Vortex3	1954	7.2
Cecil	932	6.5
Dylan	925	5.5
Harlequin	666	6.7
Unidraw	614	4.0
PRMcl	479	4.6
Lov-obj-ed	436	8.5
SmartEiffel	397	8.6
Total	38,784	7.3

Table II. Statistics on random class-loading orders.

103,445	Lower	PN-and+shift				PN-and				PH-and+shift			PH-and			
0.12	bound	min	μ	σ	max	min	μ	σ	max	min	μ	max	ref	min	μ	max
IBM-SF	1.42	2.0	3.8	1.5	*61.2*	3.4	8.5	5.4	*120.3*	3.3	6.6	84.8	10.4	9.4	*19.3*	221.3
JDK1.3.1	1.24	1.3	1.6	0.2	*23.0*	1.8	3.4	1.9	*57.0*	1.8	2.4	27.0	11.7	6.4	*10.6*	239.1
Java.1.6	1.27	1.4	1.6	0.2	*27.3*	1.9	3.6	1.8	*52.9*	1.8	2.3	26.1	8.0	5.9	*10.5*	163.3
Orbix	1.13	1.1	1.3	0.1	*15.5*	1.2	1.9	1.1	*34.2*	1.3	1.7	33.0	4.4	4.0	*6.0*	103.1
Corba	1.19	1.3	1.8	0.4	*24.5*	1.6	2.9	1.4	*41.0*	1.7	2.7	67.4	5.1	4.4	*7.2*	93.8
Orbacus	1.20	1.4	2.0	0.5	*18.2*	1.9	4.0	1.9	*43.3*	2.0	3.2	36.0	5.3	4.9	*8.7*	52.4
HotJava	1.31	1.4	2.1	0.7	*24.4*	1.7	3.5	1.8	*33.9*	2.1	3.7	33.2	6.4	4.4	*8.1*	34.9
JDK.1.0.2	1.30	1.3	1.4	0.2	*10.8*	1.3	1.8	0.8	*12.2*	1.7	2.9	21.4	7.4	3.8	*7.1*	36.6
Self	1.30	1.9	3.9	0.5	8.2	2.1	4.0	0.2	8.2	5.5	9.2	21.2	5.9	5.8	9.2	21.2
Geode	1.48	2.9	6.4	2.0	*31.0*	3.6	*8.9*	2.8	*32.3*	5.2	10.6	34.8	11.5	*7.4*	*14.9*	40.3
Vortex3	1.33	1.7	2.7	0.7	*15.7*	2.3	4.7	1.4	*18.0*	3.0	5.0	27.9	11.0	6.9	*12.2*	64.0
Cecil	1.28	1.5	2.4	0.6	*12.2*	1.7	3.7	1.3	*18.2*	2.5	4.4	33.3	8.3	5.3	*9.4*	37.0
Dylan	1.35	1.4	1.6	0.4	*9.6*	1.4	1.6	0.4	*9.6*	4.2	6.9	36.0	4.6	4.2	*6.9*	36.0
Harlequin	1.32	1.8	2.9	0.5	*10.1*	2.2	4.0	0.8	*11.0*	2.9	4.9	14.7	5.9	5.1	*9.0*	23.3
Unidraw	1.27	1.3	1.3	0.0	**2.9**	1.3	1.3	0.1	**3.1**	1.6	2.4	21.2	4.2	**3.7**	6.1	34.8
PRMcl	1.31	1.3	1.5	0.2	*8.2*	1.4	2.0	0.6	*12.5*	1.8	2.7	11.2	4.4	3.8	*6.3*	22.1
Lov-obj-ed	1.38	2.2	3.8	0.8	*12.6*	2.8	*5.2*	1.1	*13.4*	3.5	6.0	14.2	6.3	*5.1*	*8.7*	17.9
SmartEiffel	1.41	1.4	1.9	0.9	*12.2*	1.4	1.9	0.9	*12.2*	4.2	7.0	17.2	4.6	4.2	*7.0*	17.2
Total	1.33	1.8	3.1		*30.0*	2.4	5.4		*55.2*	3.4	5.9	43.9	8.6	6.8	*12.6*	125.7

The top-left numbers represent, respectively, the total sample count (i.e. the number of class-loading orders that have been generated at random) and an estimation of the growth rate of the maximum values, expressed as the number of new maximal records per thousand. The first column presents the lower bound for bit-wise hashing, as a ratio $\sum_c 2^{\lceil \log_2(n_c) \rceil} / \sum_c n_c \in [1,2[$. All other numbers are ratios $\rho = \sum_c H_c / \sum_c n_c$, whereby the sum is obtained for all classes, H_c is the hashtable size and the denominator is the cardinality of $\preceq$. The minimum, average (μ) and maximum values of ρ are presented for each technique, and the standard deviation σ is also displayed for PN-and and PN-and+shift. The 'PH-and ref' column recalls the tests presented in PHAPST. Italic type represents 'bad' PN behavior, namely PN avg or max that are respectively greater than the corresponding PH min or avg. In contrast, bold type represents 'good' behavior, namely PN max that are less than PH min. Line 'Total' represents, for all columns, the same ratios as for each benchmark, but computed from the sum of the corresponding parameters on all benchmarks, i.e. when a column depicts some p_b/q_b ratio for each benchmark b, the last line is $\sum_b p_b / \sum_b q_b$.

DOI: 10.1002/spe

$\sum_c 2^{\lceil \log_2(n_c) \rceil} / \sum_c n_c$ lower bound of PH-and ranges between 1 and 2 (Section 3.2). Moreover, as a hashtable occupation ratio, $\rho = 2$ ensures a good average efficiency for usual hashtables based on *linear probing* [20–22]. According to the statistics presented in [10], the n_c average is less than 10% of the average number of methods per class; hence ,$\rho = 2$ also corresponds to a hashtable size that is less than half the method table itself. Finally, a difference of 1 in the ρ value represents less than 2% of the 900 kB stripped executable of the PRMcl benchmark which is an actual program (see Section 5.2).

We tested the following variants: (i) PH-and perfect class hashing, in its original variant with consecutive class numbering; (ii) 2-parameter perfect class hashing, with PH-and+shift; (iii) PN-and perfect class numbering and (iv) 2-parameter perfect class numbering, with PN-and+shift. We also tested quasi-perfect hashing (qPH-and), but we do not present the results, as it is not better than PN-and. Table II presents the statistics of all kept variants, in a left to right order that roughly corresponds to increasing ρ ratios.

The first conclusion is that simple perfect hashing with consecutive numbering is really bad. In most benchmarks, the results presented in PHAPST (column 'PH-and ref') are close to the PH-and minimum values, hence rather optimistic. This is disappointing, since we expected some average behavior when taking an arbitrary order. Moreover, the PH-and variations are marked, as the max/min ratio is greater than 20 for the five largest benchmarks. In the worst cases, PH-and can be dramatically inefficient—ρ can be greater than 200—and it only depends on the class loading order which is a problem input.

In contrast, perfect numbering produces markedly better results. The PN-and average is always lower than the PH-and minimum, except again in a few cases; however, the PN-and maximum is generally higher than the PH-and average, although markedly lower than its maximum. PN-and and PH-and+shift yield similar results but PN-and+shift improves on both, especially in the worst cases (maximum values); however, for many benchmarks, the worst-case size remains more than twice that of the PH-and estimation in PHAPST.

These results are rather negative. Even when considering the most compact variant, i.e. PN-and+shift, worst cases (column 'max') remain over space-consuming for many benchmarks (especially IBM-SF and Geode). As PN-and and PN-and+shift are certainly the most promising techniques, we produced more statistics for them. The positive point is that their standard deviation σ is rather low. For all benchmarks but IBM-SF and Geode, σ is indeed less than 2 for PN-and, and than 1 for PN-and+shift. As an empirical verification of the Bienaymé–Chebyshev inequality, we also observed that, for each benchmark, the proportion of class-loading orders that exceed $\mu + 2\sigma$ is less than 5%, for both PN-and and PN-and+shift.

In PHAPST, we did not conclude on PN because our first results seemed erratic. Actually PH-and itself is erratic, and PN is always a marked improvement over PH. However, the technique remains space-inefficient with bit-wise-and in the worst-case class-loading orders, although these worst cases are not frequent. But now, the PN-and average is always better than the PH-and number in PHAPST. Therefore, restricting class-loading orders might make PN-and acceptable in all cases. Indeed, all linear extensions are not sensible class-loading orders. Thus, modelling class-loading orders and testing perfect hashing on 'sensible' class-loading orders are still open issues.

4.2. Random leaf-class loading

The main issue with these first tests is that the bit-wise-and hashing function appears to be highly dependent on the class-loading order. Class loading depends on precise implementation of run-time systems like virtual machines, and the class-loading order depends on the considered program behavior. However, without loss of generality, one may assume that class loading is always triggered by the need to instantiate a yet unloaded class, as all other uses of yet unloaded classes could be made lazy. Hence, only *concrete*, i.e. non-abstract, classes must be ordered and *abstract classes* are only inserted when needed in concrete-class orders, in such a way that superclasses are always loaded before subclasses.

DOI: 10.1002/spe

Table III. Statistics on class, leaf and superclass numbers.

	Class number	Leaf number	Avg. n_c
IBM-SF	8793	6001	8.8
JDK1.3.1	7401	5806	4.5
Java.1.6	5933	3825	4.8
Orbix	2716	2440	2.7
Corba	1699	1473	3.7
Orbacus	1379	954	4.7
HotJava	736	525	5.6
JDK.1.0.2	604	445	4.9
Self	1802	1134	31.9
Geode	1318	732	14.7
Vortex3	1954	1216	7.4
Cecil	932	601	6.8
Dylan	925	806	5.6
Harlequin	666	371	7.5
Unidraw	614	481	4.0
PRMcl	479	294	5.1
Lov-obj-ed	436	218	9.9
SmartEiffel	397	311	8.9
Total	38,784	27,633	7.0

4.2.1. Principle. As our benchmarks do not record whether a class is abstract or concrete, we consider an assumption that is often advocated, namely *'make all non-leaf classes abstract'* [9]. This is indeed a common methodological recommendation; see for instance [23]. Table V presents the leaf number and the average n_c per leaf for all benchmarks. On average, according to this assumption, there would be a little more than one abstract class to four classes. This is an upper bound of the actual ratio, since it is meaningless for a leaf class to be abstract, and a little more than the ratio of one to six advocated by Lorenz and Kidd [24] (however, at a time when class hierarchies were markedly smaller).

An informal algorithm is as follows. An unloaded leaf c is selected at random, and the set $X_c = \{x \mid c \preceq x\}$ is partitioned into two subsets X'_c and X''_c, that contain, respectively, already loaded and yet unloaded superclasses, including c. X''_c is then ordered in a top–down linear extension. For each class $x \in X''_c$, in this order, id_x and H_x are computed according to the considered algorithm (PH or PN). In the PN case, instead of considering each class in X''_c separately, an alternative is a global optimization of X''_c numbering, with perfect k-numbering (Definition 3.4). There are actually pros and cons for both approaches, since PN-`and` is optimal for single-inheritance classes (Proposition 3.14) and the simple optimal problem (Conjecture 3.5) has no solution in the general case. However, our experiments show that global optimization gives slightly better results.

There is however a huge number of linear extensions, namely factorial of the number of leaves, which may be as many as 6000 in the largest benchmarks (Table IV). The number of orders is thus above 10^{20000}; obviously, the point cannot be to compute exact statistics. It is possible to somewhat reduce this combinatorial explosion by partitioning the set of leaves according to their parents; indeed, the way leaves with the same parents are ordered with each other is not significant. Therefore, a two-step selection avoids equivalent orders. Sets are represented as lists, and a set of equivalent leaves is first selected at random, then the first element in this set is picked. The number of orders is now $\mathrm{fact}(\sum_{i=1}^{k} p_i) / \prod_{i=1}^{k} \mathrm{fact}(p_i)$, instead of $\mathrm{fact}(\sum_{i=1}^{k} p_i)$, whereby k is the number of equivalence classes and p_i the cardinality of each of them. In this way, the combinatorics reduces to about 10^{12000} orders; this is better but still exhaustively intractable (10^{30} ns represent a round upper bound for the solar system life). Although we were unable to prove that there were no frequent worse cases, our experiments tended to quickly converge, that is, after some thousands of tests, the rate of new maximal records was rather low, i.e. less than one to a thousand, the average remained stable and the growth of the maximum extremely slow. Leaf partition is also used for the statistics in Table II.

DOI: 10.1002/spe

Table IV. Statistics on random concrete-class loading orders.

136, 170 0.16	Lower bound	Useful PN-and				PN-and				PH-and+shift			PH-and		
		min	μ	σ	max	min	μ	σ	max	min	μ	max	min	μ	max
IBM-SF	1.43	1.6	2.0	0.2	4.2	2.0	3.4	0.6	9.6	6.8	9.2	20.3	7.4	9.9	57.9
JDK1.3.1	1.24	1.3	1.4	0.1	2.0	1.5	2.1	0.4	8.1	3.7	5.6	22.0	5.9	8.3	41.2
Java.1.6	1.29	1.3	1.5	0.1	2.1	1.5	2.3	0.4	*10.9*	4.1	6.5	21.9	5.8	*8.5*	24.6
Orbix	**1.11**	**1.1**	1.2	0.0	1.3	1.1	1.3	0.2	*6.6*	1.6	2.3	8.9	3.8	*4.8*	36.6
Corba	1.18	1.2	1.3	0.1	1.8	1.2	2.0	0.7	*9.8*	2.4	3.7	12.1	4.0	*5.5*	31.4
Orbacus	1.20	1.2	1.5	0.2	3.1	1.2	2.1	0.8	*15.5*	3.2	5.2	18.5	4.3	*6.5*	30.6
HotJava	1.33	1.3	1.6	0.1	3.0	1.4	2.0	0.3	5.0	3.3	5.0	8.8	4.1	6.0	11.4
JDK.1.0.2	**1.32**	**1.3**	1.4	0.1	2.6	**1.3**	1.5	0.2	4.8	2.6	4.7	10.4	3.6	5.7	11.9
Self	1.33	1.3	1.4	0.1	2.8	1.3	1.5	0.1	**2.8**	4.9	6.1	13.5	**4.9**	6.1	13.5
Geode	1.48	1.7	2.2	0.3	5.9	1.9	2.9	0.8	*15.0*	5.1	7.9	26.3	5.2	*8.1*	26.5
Vortex3	1.33	1.4	1.8	0.2	4.1	1.5	2.5	0.4	6.5	5.0	7.7	14.7	5.6	8.3	15.0
Cecil	1.27	1.3	1.6	0.2	3.6	1.4	2.2	0.4	5.5	4.1	6.4	12.9	4.5	7.0	13.9
Dylan	**1.37**	**1.4**	1.4	0.1	1.8	**1.4**	1.7	0.6	*15.1*	3.8	5.7	22.6	3.8	*5.7*	22.6
Harlequin	1.33	1.5	1.9	0.2	3.3	1.5	2.3	0.3	5.5	3.9	6.3	12.3	4.4	6.8	13.2
Unidraw	**1.27**	**1.3**	1.3	0.0	1.4	**1.3**	1.3	0.0	**1.9**	2.3	3.6	7.1	**3.5**	4.9	11.4
PRMcl	**1.32**	**1.3**	1.4	0.1	2.0	**1.3**	1.6	0.2	*5.7*	2.8	4.5	11.4	3.5	*5.4*	12.7
Lov-obj-ed	1.41	1.5	2.0	0.2	3.4	1.6	2.6	0.4	5.2	4.0	5.9	9.4	4.1	6.1	9.6
SmartEiffel	**1.42**	**1.4**	1.4	0.0	2.0	**1.4**	1.5	0.1	**2.9**	4.0	5.2	8.0	**4.0**	5.2	8.0
Total	1.34	1.4	1.7		3.1	1.6	2.3		*8.0*	4.8	6.8	17.8	5.7	*7.8*	32.8

All numbers have the same meaning as in Table II, but $\sum_c$ sums are now restricted to leaf classes, and the useful part of PN-and is presented instead of PN-and+shift. In the lower bound and PN 'min' columns, bold type also indicates that the optimum value is reached.

4.2.2. Results and discussion. We tested all perfect hashing functions on the same set of benchmarks, under this new assumption. We did this under two forms, according to whether, in the ratio $\sum_c H_c / \sum_c n_c$, sums are still applied to all classes or only to concrete classes. Indeed, only all-class sums are comparable to the statistics in Table II, but only concrete classes require method tables. As the resulting ratios do not markedly differ between the two forms, we only present the second one. Furthermore, as PN-and+shift is now hardly better than PN-and, we present statistics on the useful part of the PN-and tables according to Proposition 3.11.

In this setting, all the PH-and values are markedly improved (Table IV), and the PHAPST ratios in Table II are now close to the average PH ratios in Table IV. This is explained by the fact that our original class order was a depth-first top–down linear extension, hence a possible leaf-class ordering. The fact that the minimum values are greatly improved confirms that our random testbed cannot explore all possible orders (the combinatorics is too large), whereas leaf-class ordering focuses on the best orders. Regarding PN, it is not clear whether the optimization takes advantage of loading and numbering a set of classes as a whole. We tested 1- and k-numbering; on average, the resulting H_c parameter does not significantly differ, regardless of whether all yet unloaded classes are numbered as a whole (k-numbering), or whether they are numbered one by one by successive applications of PN in a top–down ordering (1-numbering). Hence, the improvement over PH is mostly due to the selection of specific class-loading orders.

The most interesting observation is that the erratic behavior of PH-and has been markedly reduced (the max/min ratio is now always less than 10, instead of 35) and most worst-case orders have been ruled out (the worst-case ratio ρ hardly exceeds 50, instead of 200). On most benchmarks, the maximum PN-and values in Table IV are now lower than the corresponding PH-and minimal values in Table II. The standard deviation is now less than 1 on all benchmarks; the proportion of leaf-class orders that exceed $\mu+2\sigma$ is still less than 5%; and $\mu+2\sigma$ is always less than 5. Moreover, for the useful PN-and ratio, $\mu+2\sigma$ is always less than 3, and the maximal values are markedly lower than that of PN-and. Hence, a large part of the extra memory occupation is not wasted but can be reused for specific static allocations. Overall, PN-and produces excellent results. On average,

DOI: 10.1002/spe

Table V. PN-and statistics on a subset of all classes, with random leaf-class-loading orders.

Fraction sample size	60% 246,680				40% 237,830				30% 230,690				20% 230,690			
	min	μ	σ	max	min	μ	σ	max	min	μ	σ	max	min	μ	σ	max
IBM-SF	2.1	3.2	0.4	5.9	1.8	2.8	0.3	5.7	1.8	2.6	0.3	6.6	1.7	2.4	0.3	5.0
JDK1.3.1	1.6	2.4	0.4	6.4	1.6	2.5	0.4	6.7	1.5	2.3	0.4	6.5	1.4	2.1	0.3	6.2
Java.1.6	1.5	2.3	0.4	5.9	1.4	2.2	0.4	6.0	1.4	2.0	0.3	5.7	1.3	1.9	0.3	5.0
Orbix	1.2	1.4	0.3	9.5	1.2	1.5	0.3	13.0	1.2	1.5	0.4	14.6	1.2	1.6	0.2	3.7
Corba	1.3	2.3	1.0	11.8	1.3	2.4	0.8	11.4	1.3	2.3	0.5	6.7	1.3	2.1	0.3	4.0
Orbacus	1.3	2.1	0.5	7.9	1.3	1.9	0.3	4.6	1.3	1.8	0.3	4.9	1.2	1.6	0.2	3.8
Self	1.4	1.6	0.1	3.4	1.4	1.5	0.1	3.3	1.4	1.5	0.1	3.3	1.3	1.5	0.1	3.5
Geode	2.1	3.1	0.4	5.9	1.8	2.9	0.4	5.7	1.6	2.6	0.4	5.6	1.4	2.3	0.3	4.5
Vortex3	1.6	2.5	0.4	6.7	1.5	2.3	0.4	6.1	1.4	2.1	0.4	5.4	1.3	1.9	0.3	5.8
Cecil	1.4	2.1	0.4	5.5	1.3	1.9	0.3	5.2	1.2	1.8	0.3	5.4	1.2	1.6	0.3	3.9
Dylan	1.2	1.5	0.2	7.9	1.3	1.5	0.1	4.6	1.3	1.5	0.1	3.4	1.2	1.5	0.1	3.3
Harlequin	1.5	2.3	0.4	5.0	1.3	2.1	0.3	4.7	1.2	1.9	0.3	4.4	1.1	1.6	0.2	3.5
Total	1.5	2.2		6.8	1.4	2.1		6.4	1.4	2.0		6.1	1.3	1.8		4.3

Only a subset of all classes are now loaded, and sums are on all loaded leaf classes. Column '100%' would display the same data as column PN-and in Table IV.

its space-occupation ratio is close to 2 for most benchmarks, and perfect hashing is thus obtained at a cost similar to that of efficient linear probing. The ratio is often very close to its $2^{\lceil \log_2(n_c) \rceil}$ lower bound, although it does not exclude a risk of a bad-case class-loading order. However this risk is low, as bad cases are infrequent; and it is not fatal, as the resulting memory occupation would be large but not unreasonable. Actually, it would seem that there is no need for any other hashing function, and PN-and+shift is not sufficiently better to offset its expected time-overhead.

Of course, it would be interesting to confirm that leaf-class orders are representative of actual class-loading orders. The reality is likely midway between both models. On the one hand, programmers and class hierarchies only partly comply with Meyers' directive. On the other hand, concrete classes would represent only a possibility, since a concrete class can be instantiated in some programs, but the class may also remain abstract in many others. This concerns only non-leaf classes, since a non-instantiated leaf would never be loaded.

4.2.3. Loading a subset of all classes. However, apart from some small benchmarks such as PRMcl and SmartEiffel, our benchmarks do not represent actual programs, but whole class libraries. Therefore, a real program would only load a subset of these libraries, and it was reported in [17] that hardly 1000 classes were actually loaded from similar several-thousand class hierarchies.

Therefore, we also tested PH-and variants when only a subset of all classes was loaded. The principle is the same as random class ordering, except that class loading ends when a fraction of all classes are loaded. In practice, we tested it with fractions running from 20 to 60%. The class number limitation improves all PH parameters, but the bad features of PH-and remain, although at a smaller scale. Table V presents only the statistics of PN-and in this new setting, for the restriction to leaf classes. The table shows marked improvement as the class number decreases.

4.3. Application to JAVA*-like multiple subtyping*

Perfect hashing can also be applied to JAVA-like interfaces in a restricted form. In JAVA, the `extends` relationship between classes is in single inheritance; thus, Cohen's display and usual single-subtyping implementation works, and it is more efficient than PH for classes (Section 2.1). Therefore, only the `implements` relationship between classes and interfaces needs to be hashed. The exact relation is not the explicit `implements` relationship, but rather its closure with class specialization, i.e. if *B* `extends` *A* and *A* `implements` *I*, then *B* `implements` *I*. Overall, only classes require a hashtable and only interfaces are numbered and hashed. In this

DOI: 10.1002/spe

Table VI. Statistics on classes and interfaces for JAVA benchmarks.

	Class numbers				Total		Extends		Implements	
	All	Concrete	Abstract	Interface number	avg	max	avg	max	avg	max
Eclipse	59,690	56,117	3573	8122	5.6	36	3.5	16	2.1	28
Java-6-sun	17,213	15,777	1436	2257	4.5	29	3.1	10	1.4	20
Total	76,903	71,894	5009	10,379	5.4	36	3.4	16	2.0	28
Derby	1764	1528	236	453	5.2	20	3.1	8	2.0	14
Jfreechart-1.0.5	1393	1215	178	382	5.3	19	3.1	8	2.2	13
Jess	1289	1137	152	329	4.3	16	2.8	7	1.6	9
Sunflow	1140	982	158	283	4.3	15	2.8	7	1.5	9
Ant	1022	895	127	187	4.2	14	3.2	8	1.1	8
Xom-1.1	898	796	102	262	4.2	12	3.0	9	1.3	7
Javac	864	770	94	207	4.3	12	3.0	6	1.4	8
Tidy	793	679	114	230	4.1	11	2.8	7	1.3	7
Jl1.0	778	655	123	215	4.1	12	2.8	7	1.3	8
Janino	783	668	115	132	4.4	13	3.2	7	1.2	9
Scheme-builder	340	275	65	82	4.4	12	3.0	6	1.4	8
Check	326	266	60	75	4.3	11	3.0	6	1.4	7
Total	11,390	9866	1524	2837	4.5	20	3.0	9	1.6	14
Jython	3715	3494	221	347	6.3	14	4.4	8	1.8	8
Fop	2669	2338	331	633	4.3	27	3.1	9	1.3	21
Xalan	2562	2309	253	490	5.2	18	3.2	9	2.0	13
Pmd	1747	1548	199	397	4.7	18	3.2	8	1.5	12
Antlr	1417	1239	178	302	4.2	15	3.0	7	1.2	10
Eclipse-dacapo	872	759	113	226	3.9	14	2.7	6	1.2	8
Luindex	452	356	96	89	4.1	12	2.9	6	1.2	7
Lusearch	396	316	80	86	4.2	11	2.9	6	1.3	7
Total	13,830	12,359	1471	2570	5.0	27	3.5	9	1.6	21

From top to bottom, there are three benchmark groups: complete libraries, SPECjvm2008 and DaCapo benchmarks. The first columns represent the class and interface numbers. The next section represents the average and maximum numbers of superclasses and implemented interfaces, per concrete class. For instance, on average on the whole Java-6-sun library, a concrete class extends 3.1 classes and implements 1.4 interfaces, and the maximum values are, respectively, 10 and 20. Moreover, the maximum value of the union of both is 29. n_c is now the number of interfaces implemented by concrete class c. 'Total' lines include all benchmarks of each group.

setting, the hashtable is used for subtype tests whose target is an interface, and for method invocations when the receiver is typed by an interface and the method is not introduced by the `Object` root.

We have adapted all perfect hashing and perfect numbering functions to JAVA interfaces and tested them on new class hierarchies that have been extracted from DaCapo [25] and SPECjvm2008 (`http://www.spec.org/jvm2008/`) benchmarks, or full Java 1.6 libraries[¶]. Table VI presents the number of classes and interfaces for each benchmark, along with the average numbers of extended classes and implemented interfaces. Each benchmark represents an approximation of the set of classes that could be loaded when the corresponding archive (`.jar`) file is loaded. These sets are generated by meta-programming, and are closed under two relationships: specialization (i.e. `implements` and `extends`) and method signature. The only classes that might defy this analysis are those that are instantiated within a method while being only a strict subclass of the collected classes. These new benchmarks provide extra information about classes, namely if they are abstract, along with their nesting and visibility. Therefore, the algorithm described in Section 4.2 has been improved in order to take the new information into account. Concrete classes are considered, instead of leaf-classes, and complex loading rules are taken into account for inner

[¶]These new benchmarks are available on `http://www.lirmm.fr/~morandat/benchmarks`.

DOI: 10.1002/spe

Table VII. Statistics of PN-and for JAVA benchmarks over random concrete-class-loading orders.

1,765,731 0.01	Lower bound	PN-and			
		min	μ	σ	max
Eclipse	1.34	1.7	3.2	1.6	28.5
Java-6-sun	1.46	1.8	2.4	0.4	8.1
Total	1.36	1.8	3.0		25.3
Derby	1.37	1.4	1.7	0.2	5.5
Jfreechart-1.0.5	1.36	1.4	1.8	0.2	3.6
Jess	1.24	1.3	1.4	0.1	4.7
Sunflow	1.34	1.4	1.5	0.1	3.7
Ant	1.50	1.5	1.6	0.1	2.9
Xom-1.1	1.40	1.4	1.5	0.0	2.2
Javac	1.44	1.4	1.6	0.1	2.6
Tidy	1.34	1.3	1.4	0.1	2.6
jl1.0	1.39	1.4	1.5	0.1	2.5
Janino	1.43	1.4	1.5	0.0	2.2
Scheme-builder	1.37	1.4	1.5	0.1	2.2
Check	1.37	1.4	1.4	0.0	1.9
Total	1.37	1.4	1.6		3.6
Jython	1.28	1.3	1.4	0.4	11.8
Fop	1.45	1.5	1.7	0.1	4.1
Xalan	1.41	1.5	2.1	0.4	5.4
Pmd	1.29	1.3	1.5	0.1	5.3
Antlr	1.51	1.5	1.6	0.1	3.3
Eclipse-dacapo	1.37	1.4	1.5	0.1	2.6
Luindex	1.45	1.5	1.5	0.1	2.2
Lusearch	1.38	1.4	1.5	0.0	2.0
Total	1.37	1.4	1.7		6.9

All numbers are ratios $\rho = \sum_c H_c / \sum_c n_c$, whereby the sum is done on concrete classes only (the denominator is the cardinality of the `implements` relationship, restricted to concrete classes). As n_c can be zero, the lower bound is now $\rho = \sum_c o_c / \sum_c n_c$, where $o_c = 1$ if $n_c = 0$ and $2^{\lceil \log_2(n_c) \rceil}$ otherwise.

classes. Indeed, JAVA inner classes are archived in separate `.class` files, and they can be loaded independently of their enclosing class. However, non-static inner classes are only instantiable from their enclosing class or a subclass of it. In contrast, static inner classes can be instantiated from the enclosing class, the package or the whole hierarchy according to their visibility. Finally, inner classes can be instantiated through invocation of a virtual or static method (both of a class), or by initializing a static variable (of a class or even an interface). Overall, the enclosing class or some class of the same package must often be loaded before the considered inner class can be loaded.

The application of Definition 3.8 is now slightly different, and resembles an application to unrooted hierarchies, since `Object` is considered as a class. Moreover, a class does not encode itself in its hashtable. Thus, for a class c that does not implement any interface other than `Object`, $I_c = \emptyset$, $n_c = 0$ and $H_c = 1$, and the single table entry must then contain any non-null integer that is not an interface ID. Proposition 3.14 optimum now occurs when interfaces are in single inheritance, and with the difference that $H_c = 1$ instead of $2^{\lceil \log_2(n_c) \rceil}$ when $n_c = 0$. Therefore, the global optimum is unbounded. Finally, the PH algorithms are applied in the same manner as described in Section 4.2.1, except that X'_c and X''_c are now sets of, respectively, already loaded and still unloaded interfaces implemented by concrete class c.

Table VII presents the statistics of PN-and over these random concrete-class-loading orders. The results are clearly satisfactory. Apart from Java-6-sun and Eclipse that represent complete libraries that will never be loaded as a whole in a single program, all other benchmarks represent

DOI: 10.1002/spe

set of classes, a large part of which could be loaded in a single run. Their PN-and parameter is quite good, namely lower than 2 on average, and $\mu+2\sigma$ rarely exceeds 2. These data will be further discussed in Section 6. The overall conclusion is that PN-and would also provide a scalable implementation for JAVA-like interfaces.

5. TIME-EFFICIENCY TESTS

This section presents the results of experiments that provide empirical assessment of the PH run-time efficiency and complement the PHAPST abstract assessment.

5.1. Abstract assessment

In Driesen's [2001] computational model, the cycle count of a code sequence is a function of the processor latencies for different machine instructions like load (L latency) or branch (B latency). We add to these two parameters the integer division latency (D) for PH-mod. Besides these latencies, processors are characterized by their instruction-level parallelism. Overall, each code sequence can be associated with a cycle count that represents the time necessary to execute the sequence when all data are cached. Driesen's model is purely static. Therefore, cache misses are not considered, whereas each one would represent a latency far greater than the cycle count of the sequences considered here. Moreover, well-predicted branchings cost nothing, whereas a misprediction costs B cycles. These two dynamic features, i.e. cache misses and mispredictions, can however be considered as a risk. For instance, IC introduces extra cache-miss risks, since three memory areas are concerned, instead of one in all other techniques. Moreover, it also introduces extra misprediction risks for subtype testing, since it requires an extra bound check. There are thus two different ways of failing (instead of one). Table VIII presents the results of these analyses for the considered techniques.

5.2. Empirical run-time assessment in the PRM *testbed*

We also implemented PH in the compiler of the PRM language, and tested its time efficiency on a real program, namely the PRM compiler itself. In this setting, perfect hashing is compared with different techniques on a variety of processors. These experiments are original, as they compare different implementation techniques, in a common framework that allows a fair comparison, with *all other things being equal*. In this section, we only summarize the results of these tests. Interested readers are referred to [7] for a more complete report.

Table VIII. Cycle counts for the different techniques and three mechanisms.

Technique	Method call			Subtype test			Attribute access		
	Cycles	*Code*		Cycles	*Code*		Cycles	*Code*	
Coloring	$2L+B$	16	3	$2L+2$	8	4	L	3	1
IC	$4L+B+2$	24	8	$3L+4$	13	10	$4L+2$	14	8
PH-and	$4L+B+3$	25	8	$3L+5$	14	8	$4L+3$	15	8
PH-and+shift	$4L+B+4$	26	10	$3L+6$	15	10	$4L+4$	16	10
PH-mod	$4L+B+D+2$	49	8	$3L+D+4$	38	8	$4L+D+2$	39	8

The table recalls the cycle counts and code lengths that are presented in PHAPST, according to the computational model of [6] which is illustrated in Figures 1, 3 and 4. L, B and D represent the respective latencies of memory loads, indirect or mispredicted branches and integer division. The values considered here are $L=3$, $B=10$ and $D=25$ (instead of the optimistic value of 6 used in PHAPST). For each mechanism, all techniques present the same cache-miss or misprediction risks, except attribute access for which PH and IC add cache-miss risk, and IC which adds misprediction risk for subtype testing and cache-miss risks in all mechanisms.

DOI: 10.1002/spe

Tested techniques. The PRM testbed has been used for comparing a variety of techniques from which we extract four families:

- Link-time *coloring* (Section 2.2.2), as a generalization of single-subtyping implementation, represents a reference, i.e. the *zero* point of our scale; this is the best-known technique that does not involve global optimization, but it requires global linking; it is also the technique used in JAVA for implementing classes, hence for method invocation and subtype testing when the receiver's type or the target type is a class, and for all attribute accesses.
- Compile-time coloring, with type analysis for detecting monomorphic call sites, represents another reference, close to the optimum, and the difference between link-time and compile-time coloring can be considered as the *unit* of our scale.
- All PH variants have been tested. As the tests concern only time-efficiency, PH and PN do not differ apart from the method-table sizes, but this slight difference should not affect cache misses.
- The *incremental* version of *coloring* (IC) (Section 2.2.4) represents an alternative to PH in spite of its expected load-time overhead.

All techniques are considered for application to the three basic mechanisms required by object-oriented programming: (i) subtype tests, (ii) method invocation and (iii) attribute access. For attribute access, two variants are considered, with attribute coloring (AC) or AS. The tests with AS provide an assessment of the use of the considered technique (IC or PH) for implementing full multiple inheritance, for instance as an alternative to the subobject-based implementation of C++. AC is the most efficient implementation, that of JAVA for instance, hence combining PH or IC with AC provides an assessment of the considered techniques when applied to JAVA interfaces. Of course, this assessment requires some extrapolation, as PRM classes are not JAVA interfaces. However, this experiment is much more demanding than actual JAVA tests, since the tested technique here represents all method invocations instead of being reserved to interface-typed receivers. Therefore, if the overhead is low in the PRM testbed, it should be even lower with actual JAVA programs.

Tested program. We implemented all these techniques in the PRM compiler, which is dedicated to exhaustive assessment of various implementation techniques and compilation schemes [7, 26]. The test involves meta-programming, as the benchmark program is the compiler itself, which is written in PRM and compiles the PRMsource code into C code. The PRM compiler is actually not compatible with dynamic loading, and the code for PH or IC has been generated at link-time exactly as if it were generated at load-time, and the usual optimizations of the PRM compiler (see [26]) are deactivated. The class-load ordering does not matter since here we only consider time measurement, and the class hierarchy is small enough (compared with the largest benchmarks in Section 4) for making size variations insignificant.

The tested program makes intensive usage of the compared object-oriented mechanisms. The respective numbers of method invocations and attribute accesses are about 1.8 and 2.6 billion. The program size is also significant; it is the PRMcl benchmark in Tables I–IV. Load-time computations are not considered here, but our previous analysis shows that it is not significant for perfect hashing in both practice and theory (see Appendix A).

Time-measurement conditions. Tested variants differ only by the underlying implementation technique, with all other things being equal. The compilation testbed is deterministic, and two compilations of the same program by the same compiler executable produce exactly the same executable. Moreover, when applied to a program, two compiler variants produce exactly the same code, and it has been verified with the `diff` command on both C and binary files. Overall, the effect of memory locality should be roughly constant, apart from specific effects due to the considered techniques. The tests were performed on several processors from different families and manufacturers: Intel® Pentium™, AMD® Athlon™, SUN® UltraSPARC™ or PowerPC™. All processors run under various 32-bit versions of Unix (Linux, Solaris, BSD).

DOI: 10.1002/spe

Table IX. Comparison of execution time according to implementation techniques and processors.

Technique	AC	AS	AC	AS	AC	AS
Compile-time MC	−1.0	−0.3	−1.0	0.5	−1.0	0.2
Link-time MC	0.0	0.5	0.0	1.2	0.0	1.1
IC	0.5	1.2	1.1	3.1	0.7	2.9
PH-`and`	0.4	1.3	1.3	4.3	0.5	2.9
PH-`and+shift`	0.6	1.8	1.5	4.9	0.8	4.2
PH-`mod`	3.0	9.2	6.0	17.9	1.9	9.3
Unit (%)	19.0		12.8		10.7	
# processor	4		2		4	
Cluster	1		2		3	

Each sub-table presents the results for a cluster of processors, and each number is the unweighted mean of the corresponding measures for all processors in the considered cluster. Each row describes a method invocation and subtype testing technique, and the two columns represent the overhead vs pure coloring (link-time MC-AC), respectively with attribute coloring (AC) and accessor simulation (AS).

5.3. Results and discussion

The results presented in [7] are here abstracted in two ways (Table IX). First, the scale is changed and the unit is now the difference in percentage between link-time and compile-time colorings. Second, a few processors are added, and the whole processor set is partitioned into three clusters by a k-mean algorithm, with each processor being represented by a 10-dimension point in this new scale.

- Cluster 1 gathers processors of different families (UltraSPARC, PowerPC and Pentium) that are generally older than other processors (from 2001 to 2006). The unit is high and PH-`and` overhead is about half a unit with AC and slightly more than a unit with AS.
- Cluster 2 consists of AMD processors (from 2003 and 2006). The unit is average and all overheads are exaggerated.
- Cluster 3 consists of recent Pentiums (from 2006 to 2009). The unit is low, PH-`and` overhead is slightly higher than in cluster 1 with AC, but markedly higher with AS. Actually, cluster 3 resembles cluster 1 for AC, and cluster 2 for AS.
- PH-`and` is generally slightly better than IC with AC, not with AS, but the differences are not marked. Moreover, the extra instructions of PH-`and+shift` entails low extra overhead.
- In contrast, the overhead of PH-`mod` is markedly higher than that of PH-`and`. Coupled with the abstract cycle count, this is the reason why we decided to give up PH-`mod` and we do not present the theoretical and experimental analyses that we have undertaken.

These empirical results can be partly explained by comparing them with the theoretical predictions in Table VIII.

- For method invocation and subtype testing, PH-`and` adds a few `loads` from a memory area that is already used by the reference technique, hence without extra cache misses, plus a few 1-cycle instructions; these extra cycles represent real overhead that is, however, slight in comparison with the overall method call cost; PH-`and+shift` adds a load and a shift that are partly or even totally done in parallel, hence with slight overhead.
- PH-`mod` adds high integer division latency, commonly estimated at about 20–25 cycles [27], hence much more expensive than extra `loads`; D latency can be estimated at $D \approx 1 + F\Delta/1.8$, whereby Δ is the difference between the PH-`mod` and PH-`and` measures (in seconds), F is the processor frequency (in GHz) and 1.8 is the number of invocations (in billions). This estimation is about 25–35 cycles (as expected), or even more, for cluster 1 and 2, but less than 10 for processors in cluster 3. This is confirmed by the fact that most processors in cluster 3 use the `Radix-16` algorithm, rather than `Radix-4` which was used in older Pentiums [28].
- AS replaces the single coloring `load` by a sequence that adds several `loads` from a memory area (i.e. the method table) that was not already used—hence, it increases the cache-miss risks with marked overhead. This overhead increases with the cost of the underlying method

DOI: 10.1002/spe

invocation technique, from method coloring to PH-and and PH-mod. The measures provide the same kind of estimation of D at $1 + F\Delta/4.3$, whereby 4.3 is the total number of mechanism invocations, in billions. On all processors, the estimation is slightly higher than the previous one by a few cycles.

- In Table VIII, the difference between IC and PH-and is not significant, and the tests confirm it, as extra indirections in IC and extra computations in PH-and cause similar overheads. However, IC implies access to extra memory areas, a first one for the color and a second one for the recomputable color-table; hence, extra cache misses are expected, which are likely underestimated in our tests. However, we have no explanation for the fact that IC is slightly better than PH-and with AS, whereas the relation is inverted with AC (e.g. in cluster 1). Overall, as the PH load-time cost is also far lower than that of IC, PH-and must be preferred to IC.

Overall, the results of these tests are quite satisfactory, and PH-and is even better than expected for method invocation and subtype testing on many processors. It should thus provide very high efficiency for implementing interfaces in JAVA and .NET. When also used for attribute access, the overhead becomes less reasonable, and the results of PH-and with AS can actually be considered as bad. However, preliminary tests with the C++-like subobject-based implementation (Section 2.2.1) show that PH-and with AS is not less efficient than subobjects [29].

Of course, the validity of such experiments that rely on a single program may be questioned. This single-benchmark limitation is inherent to our experimentation. The PRM compiler is the only one that proposes such versatility in the basic implementation of object-oriented programs. The compensation is that the language has been developed with this single goal in mind, and its compiler is the only large-scale program written in PRM. This is however a large, fully object-oriented program, which intensively uses the basic mechanisms that are tested. Moreover, the experiments compare two implementations with all other things being equal and, apart from IC, these implementations present similar processor cache-miss risks. With IC, the effect of cache misses is likely underestimated, since all color-tables are here allocated in the same memory area, whereas load-time recomputations should scatter them in the heap. Therefore, when the differences are marked, their sign should hold for all programs and only the order of magnitude should vary. In contrast, the comparison between IC and PH cannot conclude, because differences are rather low on most processors, and increased cache-miss risks might penalize IC. However, we prefer PH because of their respective load-time behaviors.

6. RELATED WORKS

Perfect class hashing is proposed here as a potential alternative to the implementation approaches that are currently undertaken in C++ and JAVA-like languages. The key features of these languages are static typing, multiple inheritance (possibly reduced to multiple subtyping) and dynamic loading or linking. For these two language families, there is some evidence that their implementations do not meet our constant-time, linear-space and inlining criteria. The original subobject-based implementation of C++ has been briefly reviewed in Section 2.2.1. It yields marked overhead, with many compiler-generated pointers in the object layout and pointer adjustments at runtime. Moreover, before perfect hashing, there were no known constant-time subtype tests for this implementation. From the memory occupation standpoint, subobjects present a cubic worst case, and large-scale experiments show that the spatial cost can be markedly higher than for all other techniques [10]. Moreover, the aforementioned comparison between PH-and and subobjects show that they provide similar time-efficiency [29]. The remainder of this section will now consider the case of JAVA and .NET languages.

6.1. JAVA*-like run-time systems*

Modern run-time systems are dedicated to languages such as JAVA and C# that use only a limited form of multiple inheritance, namely multiple subtyping of interfaces. In this setting, the implementation of classes is that of single-subtyping, and interfaces are usually implemented in

DOI: 10.1002/spe

```
// method invocation
load [object+#tableOffset], table
load [table+#cacheMethId_i], id
comp id, #targetId
beq #hit
// cache miss
store #targetId, [table+#cacheMethId_i]
...    // usual sequence of PH or IC
...    // (4-5 instructions)
...    // or any ad hoc technique
store itable, [table+#cacheMethOffset_i]
jump #end
hit: // cache hit
load [table+#cacheMethOffset_i], itable
end:
load [itable+#methodOffset], method
call method
```

```
// subtype testing
load [object+#tableOffset], table
load [table+#cacheTypeId_i], id
comp id, #targetId
beq #succeed
...    // usual sequence of PH or IC
...    // (6-8 instructions)
...    // or any ad hoc technique
store #targetId, [table+#cacheTypeId_i]
succeed:
```

The cycle count of method invocation is here $4L + B + 2$ in case of cache hit. However, the third `load` could be moved just after the second one, and run in parallel, with $3L + B + 2$ cycles. This might, however, degrade the cache-miss case.

Figure 6. Separate caches in method tables.

rather *ad hoc* ways. Currently, almost all known JAVA interface implementations do not meet one of two requirements for constant-time or incrementality. For instance, the proposal of [17] is not inherently incremental; hence, it yields potentially high load-time overhead, together with extra run-time indirections. All other techniques reported in [2, 17, 30, 31] are not time-constant. The only exception might be the proposal of using large direct access tables in SableVM [32], but it does not meet the linear-space requirement. However, the empty slots of these huge tables are used for allocating other data, and we do not know the extent to which it offsets the nonlinear size. The same trick is possible for PH-`and` according to Proposition 3.11. However, in SableVM, it cannot be used for subtype testing, for which empty entries represent failure, i.e. useful information. Therefore, the total occupied size is linear in the product of class and interface numbers.

Non-constant-time techniques generally rely on *searching* and *caching*. For instance, the interface that introduces the invoked method is cached in the method table. Then, for each method invocation, the interface ID is compared with the cache. A cache hit thus provides the right interface-table address, whereas a cache miss requires the interface ID to be searched in a data structure, before filling the cache. However, even with cache hits, the cycle count for method invocation is $3L+B+2$ (Figure 6), which is midway between PH-`and` ($4L+B+3$) and single subtyping ($2L+B$). This cache can also be used for subtyping tests, as in [31] where the underlying search is naively sequential. The cycle count is now $2L+2$ with cache hits, but not lower than PH-`and` with cache misses. As for all cache-based techniques, the resulting efficiency depends on the actual cache-hit rate. For instance, empirical cache-hit rates between 50 and 99% are reported in [17] for subtype testing. This rate can be improved with multiple caches, whereby each interface is statically assigned to a specific cache. We tested this possibly original approach in the PRM testbed with quadruple caches and an overall cache-hit rate about 80%. The experiments show that this improves only on very inefficient underlying techniques, that is PH-`mod`, on all processors but cluster 3 [7]. Moreover, caches markedly increase memory occupation in both method tables and code sequences.

An alternative involves *polymorphic inline caches* (PIC) [33], which consist of testing the receiver's dynamic type against a few expected types. It yields a series of conditional branchings. Under the CWA, all types can be exhausted, and this is called *binary tree dispatch* (BTD) [34, 35]. Experiments in the PRM testbed show that the best techniques involve *coloring* for highly polymorphic sites and BTD for other sites [7]. However, under the OWA, all types cannot be exhausted, and the code sequence must include a case for cache misses, e.g. PH. The resulting efficiency still depends on the cache-miss rate, hence on the way the expected types are computed.

Overall, the reasons for the apparent efficiency of these run-time systems must be searched elsewhere than in the used implementation technique. A first reason is that, in JAVA-like languages, the implementation question concerns only interfaces, i.e. a subset of all classes, instead of all

DOI: 10.1002/spe

classes as in PRM. Hence, cached entities are fewer and cache-hit rates higher. This is explicit in Tables II and VII, where the average number of implemented interfaces is seldom greater than 2, whereas the average number of superclasses is about 7.

However, the main reason for this apparent efficiency likely follows from the implied compilation scheme, namely *just-in-time* (JIT) compilers, rather than from the used implementation techniques. Indeed, JIT compilers rely on a provisional *closed-world assumption* that allows the compiler to perform many optimizations that are currently valid. For instance, a method call can be handled as a static call, or as a class invocation (instead of an interface invocation), because this optimization is possible in the current state of the system. Later on the compilation of some new code may force the compiler to recompile the optimized code because the underlying optimizations are no longer valid. With this approach, the run-time system must first warm up before reaching its cruising speed whereby the code is optimized and no further recompilation is required.

6.2. *Comparison with perfect hashing*

In the Java benchmarks in Table VII, abstract classes are known and the statistics represent the real memory occupation of a program that would load exactly those classes and interfaces. Of course, there is no program that would load the complete Java-6-sun or Eclipse libraries, but these benchmarks show insight into possible material limitations. For instance, in the Eclipse benchmark, the maximum number of superclasses, i.e. the longest inheritance chain, is 16, the maximum number of implemented interfaces is 28 and the maximum number of extended classes or implemented interfaces is 36. This means that there are programs that require an efficient implementation for these specific cases.

With a midsize benchmark like Derby, the PN-`and` ρ ratio (i.e. $\mu+2\sigma$) is not greater than 2.1 for 93% of concrete-class-loading orders. As the average number of implemented interfaces is 2.1, this means that the entire interface implementation requires about 4.4 (i.e. 2.1×2.1) entries per class plus a word for the H_c parameter. Each entry is two-fold, and this amounts to 10 words per class. Moreover, the average number of superclasses is 3.1. This is the average size of Cohen's display. Overall, about 13 words per class allow for both subtype tests and interfaces. When considering only subtype testing, 9 words are enough.

It is worthwhile comparing these concrete cases with the technique used for subtyping tests in the HotSpot virtual machine [31]. The proposed technique involves a data structure with three words for the cache and eight words for fixed-size Cohen's display, whereas the tested benchmarks do not require more than five words for Cohen's display (eight words are proposed for the sake of 'robustness'). Moreover, a so-called 'secondary list' is used for implementing interface cache misses and extra superclasses that exceed Cohen's display fixed size (this is what we call the 'underlying implementation'), but it is neither precisely described, nor counted in these 11 words. Overall, the proposed technique is not better than PH on midsize benchmarks like Derby, and it cannot take actual Java hierarchies into account, as the class inheritance chains can be markedly longer than 5 or even 8, and cache misses would make the technique quite inefficient in the case of Eclipse classes with 36 superclasses and implemented interfaces. The length of the 'secondary list' would be 28, and sequential search would be painful. Actually, it would seem that the technique is only tailored to the SPECjvm98 (`http://www.spec.org/jvm98/`) benchmarks, which are markedly smaller than SPECjvm2008 and DaCapo, without considering the need to scale up gracefully on larger programs.

Both techniques could be compacted by using 16-bit integers, but our experiments with the PRM testbed show that the gain in method-table size is offset by a slight increase in code size. Therefore, it should be considered only for processors that are equipped with specific half-word instructions. Overall, we consider that perfect hashing ensures several advantages over the technique proposed in [31]. PN-`and` is indeed constant-time, inlinable, scalable, and it addresses both subtype testing and method invocation mechanisms.

Another comparison can be made with the technique proposed for method invocation in JikesRVM (formerly Jalapeño) [2]. Method identifiers are hashed in a fixed-size hashtable whose entries contain addresses of stub functions that handle collisions. This article presents benchmarks

DOI: 10.1002/spe

with two values, namely 5 and 40, for the hashtable size. In JikesRVM 3.1.0, this value is 29. The first value is very low, and certainly too low to be time-efficient. In contrast, 29 and 40 are markedly higher than PH average.

Finally, it is quite hard to compare PH and PIC. Indeed, in the best cases (no cache misses) PIC is far better, but in the worst cases (no cache hits), the winner is PH. There are actually several points against PIC: (i) before JIT compilation, profiling is needed to identify the frequent types that must be expected; (ii) this profiling must be continued in the compiled code if the code is intended to remain optimized in case of a change of frequent types; (iii) the code sequences are markedly longer. It is likely that (ii) significantly degrades the efficiency. Overall, PIC makes the compilation highly complicated in contrast with the simplicity of PH.

7. CONCLUSIONS AND PROSPECTS

Our previous works on perfect class hashing [3] concluded that the technique was promising and deserved further consideration. However, a more in-depth assessment was also required. The tests presented here now allow us to draw some new and much more definitive conclusions about the time and space efficiency of perfect class hashing.

To our knowledge, PH is, together with C++ subobject-based implementation, the only constant-time technique for method invocation that allows for both multiple inheritance and dynamic loading at reasonable spatial cost. Furthermore, its space requirements are roughly linear in the cardinality of the specialization relation and far lower than that of subobjects. PH is also the only constant-time and linear-space technique for subtyping tests that allows for both multiple inheritance and dynamic loading. These assertions hold when PH is used under the perfect class numbering variant, with both `mod` and `and` hashing functions. However, the time constant of `mod` is too high, even on recent processors that use the `Radix-16` division algorithm. Therefore, PH-`mod` should be reserved for processors with very efficient integer division, that is if any exists. In contrast, the time efficiency of PH-`and` is better than expected when used for method invocation and subtyping tests. Its overhead vs coloring is not insignificant, but it seems to be the price to be payed for implementing interfaces when they are intensively used, and the tested alternatives like caching are actually markedly less scalable. PN-`and` is thus certainly a very good solution for implementing JAVA interfaces—this is actually the best solution that we are aware of. PN-`and+shift` could also be envisaged from the time standpoint, but it does not represent a neat spatial improvement on PN-`and` when leaves are assumed to be the only concrete classes.

The main contribution of this article is, however, from the space standpoint. We have proposed and tested, here, (i) an optimized approach, namely perfect class numbering; (ii) a more systematic testbed based on random class loading and (iii) a more realistic model based on concrete leaf classes. It follows from these new tests that PH-`and` is inherently erratic and over space-consuming in the worst-case class-loading orders. *Perfect class numbering* provides a marked improvement over plain perfect hashing, with both `mod` and `and` hashing functions. However, in spite of its optimality in single inheritance, it is still over space-consuming in the worst cases, but combining it with `shift` provides an improvement that is slight on average but marked in the worst cases. Finally, the assumption that all non-leaf classes are abstract makes the behavior of PH and PN markedly more regular; in practice, the PN-`and` worst cases are now infrequent and they yield reasonable memory overhead. Furthermore, a large part of the extra memory occupation could be used for allocating static data.

Moreover, when it is also used for attribute access, the perfect hashing overhead becomes unreasonable on many processors—hence, it might not be a direct alternative to the C++ subobject-based implementation.

The prospects of this work are manifold:

- The first experiments in PRM led us to expect that PH-`and` should be very efficient for implementing JAVA interfaces—thus experiments in a production virtual machine represent the next step to confirm it. Regarding run-time systems and JIT compilers, the point at issue

DOI: 10.1002/spe

cannot be to substitute perfect hashing for both caching and JIT compilers. The proposal is rather to use perfect hashing, i.e. PN-`and`, as the single underlying technique for implementing interfaces, and to adapt current JIT compilers to this technique. The observed efficiency should not be drastically changed, but our claim is that the resulting implementation should be both simple and scalable. Constant time is also an essential argument for real-time systems. This simple goal is, however, not so easy to meet. Indeed, to be efficient, perfect hashing requires a very simple but uncommon method-table layout, namely with bidirectionality and per interface grouping. Moreover, an efficient implementation of Cohen's display, without bound checks, requires allocation of method tables in dedicated areas [3]. For instance, we tried to implement PH in JikesRVM [2, 30], but we gave up because the VM overall implementation does not meet these requirements, and the expected improvement would be offset by unnecessary indirections and bound checks.

- Experiments with the PRM testbed must be generalized to other processor families, manufacturers or architectures—especially for testing more efficient integer divisions and 64-bit implementations; other programs than the PRM compiler would also be of great interest.
- Full-multiple-inheritance languages do not provide such large and rich benchmarks as JAVA-like languages, and information about concrete classes is generally not available. Therefore, our experiments in full multiple inheritance rely on the assumption that only the leaf ordering matters. This assumption should be confirmed either empirically or theoretically. However, empirical experiments are usually at a smaller scale than simulations. For instance, in [17], a report on experiments on JAVA programs based on almost 10 000 class libraries shows that each run hardly loads one thousand classes and interfaces. Bit-wise-`and` perfect class hashing and numbering are always very good on 1000 class benchmarks. It would thus seem that the scalability of these techniques can be proven only by simulations or theoretical means. This is, however, somewhat unavoidable, since scalability is the ability to gracefully scale up over future programs that do not yet exist. Conversely, empirical experiments, in spite of their smaller scale, could provide models of class-load orderings that would be less theoretical than the slogan *'make all non-leaf classes abstract'*.

As PN-`and` provides a technique for implementing all three basic object-oriented mechanisms—namely method invocation, subtype testing and attribute access—that is efficient at least for the first two, it is essential to compare PH with the C++ subobject-based technique. Both techniques are indeed the only ones that ensure constant-time method invocation in a multiple inheritance and dynamic loading framework, while being inherently incremental. We are currently conducting these tests in the PRM testbed, and our aforementioned preliminary results show that perfect hashing coupled with AC is much more efficient than subobjects [29]. However, this combination of PH and coloring is not compatible with dynamic loading, and the compatible combination, with AS instead of coloring, is not better than subobjects. The tests must be completed for precise comparisons and firm conclusions, but the overall conclusion is that both implementations are not that efficient. In practice, the efficiency of the C++ language comes from the fact that virtual inheritance is only marginal in C++ programs.

Despite this apparent inefficiency, an efficient solution to full multiple inheritance implementation in a dynamic loading setting might be based on perfect hashing, AS and JIT optimizations, and there is some evidence that such a solution could be markedly more efficient than current subobject-based implementations. We now envisage designing a virtual machine for the PRM language, with a JIT compiler adapted to full multiple inheritance and perfect hashing. In this context, a basic JIT optimization would be to use the single-subtyping implementation instead of perfect hashing until it becomes impossible, then to recompile the concerned methods. Preliminary simulations show that the single-subtyping implementation could be used on most attribute accesses, and on many method invocations, thus making a full multiple-inheritance virtual machine as efficient as a JAVA-like program making intensive use of interfaces [36]. Moreover, the recompilation cost would remain markedly low.

An extra contribution of this article concerns the methodology used for simulating load-time effects at random on very-large-scale hierarchies (Section 4). This approach is currently used to

DOI: 10.1002/spe

simulate the effects of alternative decisions in the design of the PRM virtual machine. It provides a preliminary assessment of recompilation costs and overall expected efficiency in the worst case. Such random simulations could also be used for assessing the scalability of many optimizations currently used in virtual machines and JIT compilers.

APPENDIX A: ALGORITHMS

The computation of perfect hashing parameters is presented in a simple COMMON LISP code [37], as in PHAPST. In the following algorithms, simplicity takes priority over efficiency.

A.1. Perfect class hashing with bit-wise-and

PH-and could be based on a function that takes successive numbers until it finds a perfect hashing function. This works well with modulus but, with bit-wise and, the PH parameter closely depends on the significant bits of the input numbers. Indeed, with bit-wise-and, a hashtable can have empty entries that cannot be filled by any number, and the useful entries depend only on the 1-bits of the mask (Proposition 3.7). Therefore, an algorithm working at the bit level is certainly preferable. By the way, it is also more efficient as it yields a logarithmic complexity.

The basic version of PH-and first computes a mask composed of all discriminant bits, i.e. bits that are not 0 or 1 in all numbers. The resulting integer gives a perfect hashing function since all integer pairs in the input differ by at least one bit in the mask. Then the function checks each 1-value bit by decreasing weight and switches the bit when it is not required for injectivity.

```
defun ph-and (ln)                    ;; LN is an integer list
  (if (null (cdr ln))
      1
    (let ((mask (logxor (apply #'logior ln) (apply #'logand ln))))
                                     ;; MASK consists of all discriminant bits
      (fill *ht* nil :start 0 :end (1+ mask))          ;; resets *HT*
      (loop for b from (highest-bit mask) by 1 downto 0
            when (logbitp b mask) do          ;; for each 1-bit in MASK
            (let ((new (logxor mask (ash 1 b)))) ;; NEW = MASK with switched bit
              (when (ph-and-p ln new)  (setf mask new)))
            finally return (1+ mask)))))
```

In this code, logxor, logior and logand are COMMON LISP integer functions for bit-wise operations: exclusive and inclusive or, and and. (highest-bit n) gives the position of the leftmost 1-bit of a positive integer n. (logbitp n b) tests if the bth bit of n is 1. (ash n b) shifts n left by b positions (when b is positive).

```
defun ph-and-p (ln mask)
  (loop for i in ln
        for hv = (logand i mask) do
        (if (eql (aref *ht* hv) mask)
            (return nil)
          (setf (aref *ht* hv) mask))
        finally return t))
```

The ph-and-p function checks that its mask parameter forms a perfect hashing function for the ln identifier list. *ht* is a global variable bound to an array that is presumed to be large enough and is initialized only once in ph-and (this is a slight improvement over the PHAPST version). A successful call to ph-and-p leaves the hashtable filled by numbers in ln on length hc. However, ph-and often finishes with an unsuccessful call to ph-and-p. Therefore a subsequent call to ph-and-p may be required in the following functions, in order to reset the hashtable.

With consecutive numbering of class IDs, the ph-and complexity is $\mathcal{O}(id_c + n_c \log id_c)$. Indeed, id_c is the maximum of ln input, and mask is bounded by $2id_c$ (Proposition 3.9). The *ht*

DOI: 10.1002/spe

initialization is thus in $\mathcal{O}(id_c)$, and the main loop involves $\mathcal{O}(\log id_c)$ calls to `ph-and-p`, which itself is in $\mathcal{O}(n_c)$.

A.2. Perfect class numbering

PN-`and` is slightly more complicated than PH-`and`. The algorithm is based on Proposition 3.13 as follows. It first computes the PH-`and` parameter for the input list of numbers. If the bit-wise `mask` does not contain enough 1-bits (`logcount` is the function that returns the 1-bit count of its argument), the rightmost 0-bit is switched. Then `*ht*` is reset by `ph-and-p` and the function ends by selecting free identifiers. It returns two values[‖], namely the set `idc` of new identifiers and the hashtable size `hc`.

```
defun pn-and* (ln n)
  (let ((mask (1- (ph-and ln))))
    (loop for b from 0 by 1
          while (> (+ (length ln) n) (ash 1 (logcount mask)))
                                          ;; when there are not enough 1-bits
          unless (logbitp b mask)                ;; rightmost 0-bit
          do (setf mask (logxor mask (ash 1 b))))          ;; switch
    (ph-and-p ln mask)                                ;; resets *HT*
    (values (compute-least-free-ids-and n mask) (1+ mask))))

defun compute-least-free-ids-and (n mask)
  (loop for i in *free* until (= n 0)
        for hv =  (logand i mask)
        unless (eql (aref *ht* hv) mask)
        collect (progn (setf (aref *ht* hv) mask)
                       (decf n)
                       i)))
```

`*free*` is a data structure that represents the set of free identifiers. The `loop` on `*free*` is actually slightly more complicated, as `*free*` is a more efficient data structure than a simple list, e.g. a union of intervals. It might also be optimized to take advantage of the bit-wise structure of free numbers. However, it is certainly not straightforward, and this naive loop is likely a good tradeoff between simplicity and efficiency.

Finally, the newly allocated class identifiers must be assigned to the set X_c'' of newly loaded classes, and there are many ways to do this. In this respect, `(pn-and* ln n)` is certainly not exactly equivalent to iterating `(pn-and* ln 1) n` times. Their respective results must depend, however, on the structure of the class hierarchy, and only the latter form ensures the single-inheritance optimum. Consider, for instance, the *ABCD* diamond of Figure 2. Suppose that only *A* is already numbered; hence, all other classes must be numbered as a whole (i.e. `n`=3). Proposition 3.14 states that `(pn-and ln)` is optimal in the presence of single inheritance. Hence, it should be preferred for *B* and *C*, especially if they have many subclasses. On the contrary, `(pn-and* ln n)` should likely be preferred for *D*, especially if *D* has many subclasses. Thus, the overall efficiency depends on the whole class hierarchy, which is unpredictable when the considered classes are loaded. Our experiments are actually not conclusive, as the differences are not significant.

Without considering the free-number computation, `pn-and*` adds $\mathcal{O}(\log n_c'' \log id_c')$ to the $\mathcal{O}(n_c' \log id_c')$ `ph-and` complexity, where id_c' is now the maximum inherited class ID, and n_c' and n_c'' are the respective numbers of already loaded and yet unloaded classes (i.e. n_c'' is `n`). The free-number computation is in $\mathcal{O}(id_c)$, where id_c is the maximum returned free number, which will be assigned to the current class *c*. Of course, this analysis does not provide formal complexity,

[‖]The function here uses the COMMON LISP feature called *multiple values*, with the `values` special form. Another special form that is used hereafter, i.e. `multiple-value-bind`, binds a list of variables to such multiple values.

DOI: 10.1002/spe

since it is a function of the output, instead of the input. However, as we empirically observed that the maximum allocated identifier does not exceed 3 N, where N is the total class number, this provides rough complexity on average.

A.3. Bit-wise `shift`

The combination of `and` and `shift` (Definition 3.17) relies on a similar algorithm, but it is now linear instead of logarithmic. It is indeed exponential in the 1-bit count. The function is like `ph-and` except that the 1-bits of the mask are not scanned in the same order. When both leftmost and rightmost 1-bits are switchable, both solutions are computed and the best one is returned. Otherwise, the leftmost or rightmost 1-bits are switched while it is possible, and finally the remaining inner 1-bits are switched in decreasing weight. We present a simplified version. The `best-and+s` function returns a single bit-wise mask whose rightmost 1-bit gives the parameter H_2, and H_1 is obtained by shifting the mask by H_2 positions.

```
defun ph-and+s (ln)
  (if (null (cdr ln))
      (values 1 0)                  ;; returns H_1 and H_2
    (let ((mask (logxor (apply #'logior ln) (apply #'logand ln))))
          (lbit))
        (fill *ht* nil :start 0 :end (1+ mask))                    ;; resets *HT*
        (setf mask (best-and+s mask)
              lbit (lowest-bit mask))
        (values (1+ (ash mask (- lbit))) lbit))))

(defun best-and+s (ln mask)
   (let* ((hbit (highest-bit mask))
          (lbit (lowest-bit mask))
          (newh (logxor mask (ash 1 hbit)))
          (newl (logxor mask (ash 1 lbit))))
     (if (ph-and-p ln newh)
         (if  (ph-and-p ln newl)
              (let  ((mh (best-and+s ln newh))              ;; exponential
                     (ml (best-and+s ln newl)))             ;; part
                (if  (<= (ash mh  (- (lowest-bit mh)))
                         (ash ml  (- (lowest-bit ml))))
                     mh ml))
             (best-and+s ln newh))
        (if (ph-and-p ln newl)
            (best-and+s ln newl)
          ;; switches inner 1-bits by decreasing weight
          (loop for b from (1- hbit) by 1 downto (1+ lbit)
                when (logbitp b mask) do
                (let  ((new (logxor mask (ash 1 b))))
                   (when (ph-and-p ln new)
                         (setf mask new)))
                finally return mask)))))
```

The combination with perfect numbering gives the following algorithm. The difference with respect to `pn-and*` is that the switched bit is first searched in the range between leftmost and rightmost 1-bits, if any, and otherwise the mask is extended, preferably on the right.

```
defun pn-and+s* (ln n)
  (multiple-value-bind
    (hc mask lbit hbit) (ph-and+s ln)
    (loop while (> (+ (length ln) n) (ash 1 (logcount mask))) do
          (loop for b from lbit by 1                          ;; rightmost
                unless (logbitp b mask) do                    ;; inner 0-bit
                (when (and (> b hbit) (> lbit 0))
                   (setf b (1- lbit)
                         lbit b))
                (setf mask (logxor mask (ash 1 b))          ;; switch
                      hc (1+ (ash mask (- lbit))))
                (return)))
    (ph-and-p ln mask)                                      ;; ends like PN-AND*
    (values (compute-least-free-ids-and n mask) hc)))
```


DOI: 10.1002/spe

REFERENCES

1. Lippman SB. *Inside the* C++ *Object Model*. Addison-Wesley: New York, 1996.
2. Alpern B, Cocchi A, Fink S, Grove D. Efficient implementation of Java interfaces: Invokeinterface considered harmless. *Proceedings of OOPSLA'01*. ACM: New York, SIGPLAN Notices 36(10), 2001; 108–124.
3. Ducournau R. Perfect hashing as an almost perfect subtype test. *ACM Transactions on Programming Languages and Systems* 2008; **30**(6):1–56.
4. Sprugnoli R. Perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM* 1977; **20**(11):841–850.
5. Czech ZJ, Havas G, Majewski BS. Perfect hashing. *Theoretical Computer Science* 1997; **182**(1–2):1–143.
6. Driesen K. *Efficient Polymorphic Calls*. Kluwer: Dordrecht, 2001.
7. Ducournau R, Morandat F, Privat J. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. *Proceedings of OOPSLA'09*, Leavens GT (ed.). ACM: New York, SIGPLAN Notices 44(10), 2009; 41–60.
8. Czech ZJ. Quasi-perfect hashing. *The Computer Journal* 1998; **41**:416–421.
9. Meyers S. *More Effective* C++. Addison-Wesley: Reading, MA, 1996.
10. Ducournau R. Implementing statically typed object-oriented programming languages. *ACM Computing Surveys* 2011; **43**(4).
11. Cohen NH. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems* 1991; **13**(4):626–629.
12. Ducournau R, Privat J. Metamodeling semantics of multiple inheritance. *Science of Computer Programming* 2011; DOI: 10.1016/j.scico.2010.10.006.
13. Dixon R, McKee T, Schweitzer P, Vaughan M. A fast method dispatcher for compiled languages with multiple inheritance. *Proceedings OOPSLA'89*. ACM: New York, SIGPLAN Notices 24(10), 1989; 211–214.
14. Pugh W, Weddell G. Two-directional record layout for multiple inheritance. *Proceedings of PLDI'90*. ACM: New York, SIGPLAN Notices 25(6), 1990; 85–91.
15. Vitek J, Horspool RN, Krall A. Efficient type inclusion tests. *Proceedings of OOPSLA'97*. ACM: New York, SIGPLAN Notices 32(10), 1997; 142–157.
16. Ducournau R. Coloring, a versatile technique for implementing object-oriented languages. *Software—Practice and Experience* 2011; DOI: 10.1002/spe.1022.
17. Palacz K, Vitek J. Java subtype tests in real-time. *Proceedings of ECOOP'2003* (*Lecture Notes in Computer Science*, vol. 2743), Cardelli L (ed.). Springer: Berlin, 2003; 378–404.
18. Zibin Y, Gil J. Two-dimensional bi-directional object layout. *Proceedings of ECOOP'2003* (*Lecture Notes in Computer Science*, vol. 2743), Cardelli L (ed.). Springer: Berlin, 2003; 329–350.
19. Myers A. Bidirectional object layout for separate compilation. *Proceedings of OOPSLA'95*. ACM: New York, SIGPLAN Notices 30(10), 1995; 124–139.
20. Morris R. Scatter storage techniques. *Communications of the ACM* 1968; **11**(1):38–44.
21. Knuth DE. *The Art of Computer Programming*, *Sorting and Searching*, vol. 3. Addison-Wesley: Reading, MA, 1973.
22. Vitter SJ, Flajolet Ph. Average-case analysis of algorithms and data structures. *Handbook of Theoretical Computer Science*, *Volume 1*: *Algorithms and Complexity*, Chapter 9, Van Leeuwen J (ed.). Elsevier: Amsterdam, 1990; 431–524.
23. Steimann F. Abstract class hierarchies, factories, and stable designs. *Communications of the ACM* 2000; **43**(4): 109–111.
24. Lorenz M, Kidd J. *Object-oriented Software Metrics*. Prentice-Hall: Englewood Cliffs, NJ, U.S.A., 1994.
25. Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley SK, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Moss B, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B. The DaCapo benchmarks: Java benchmarking development and analysis. *Proceedings of OOPSLA'06*, Tarr PL, Cook WR (eds.). ACM: New York, SIGPLAN Notices 41(10), 2006; 169–190.
26. Privat J, Ducournau R. Link-time static analysis for efficient separate compilation of object-oriented languages. *ACM Workshop on Program Analysis for Software Tools and Engineering* (*PASTE'05*), Lisbon, Portugal, 2005; 20–27.
27. Warren HS. *Hacker's Delight*. Addison-Wesley: Reading, MA, 2003.
28. Coke J, Baliga H, Cooray N, Gamsaragan E, Smith P, Yoon K, Abel J, Valles A. Improvements in the Intel core2 penryn processor family architecture and microarchitecture. *Intel® Technology Journal* 2008; **12**(03):179–192.
29. Morandat F, Ducournau R. Empirical assessment of C++-like implementations for multiple inheritance. *Proceedings of MASPEGHI/ICOOOLPS Workshop*. ACM: New York, 2010; 7–11.
30. Alpern B, Cocchi A, Grove D. Dynamic type checking in Jalapeño. *Proceedings of USENIX JVM'01*, Monterey, CA, 2001; 41–52.
31. Click C, Rose J. Fast subtype checking in the Hotspot JVM. *Proceedings of ACM-ISCOPE Conference on Java Grande* (*JGI'02*), Seattle, WA, 2002; 96–107.
32. Gagnon ME, Hendren LJ. SableVM: A research framework for the efficient execution of Java bytecode. *Proceedings of USENIX JVM'01*, Monterey, CA, 2001; 27–40.

33. Hölzle U, Chambers C, Ungar D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. *Proceedings of ECOOP'91* (*Lecture Notes in Computer Science*, vol. 512), America P (ed.). Springer: Berlin, 1991; 21–38.
34. Zendra O, Colnet D, Collin S. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. *Proceedings of OOPSLA'97*. ACM: New York, SIGPLAN Notices 32(10), 1997; 125–141.
35. Zendra O, Driesen K. Stress-testing control structures for dynamic dispatch in Java. *Proceedings of Java Virtual Machine Research and Technology Symposium*, *JVM'02*, Usenix: San Francisco, CA, 2002; 105–118.
36. Ducournau R, Morandat F. Towards a full multiple-inheritance virtual machine. *Proceedings of MASPEGHI/ICOOOLPS Workshop*. ACM: New York, 2010; 1–6.
37. Steele GL. *Common Lisp*, *the Language* (2nd edn). Digital Press: Bedford, MA, 1990.

DOI: 10.1002/spe