More Results on Perfect Hashing for Implementing Object-Oriented Languages

ROLAND DUCOURNAU and FLOREAL MORANDAT
LIRMM – CNRS and Université Montpellier II, France

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 paper, we have 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 from the time standpoint, and on large-scale benchmarks from the space standpoint. The conclusions were that the technique was promising but required further research in order to assess its scalability.

This article presents some new results that further highlight the benefits of this approach. We propose and test both new hashing functions and an inverted problem which amounts to selecting the best class identifiers in order to minimize the overall hashtable size. Experiments within an extended testbed with random class loading and under reasonable assumptions about what should be a sensible class loading order show that perfect hashing scales up gracefully. Furthermore, we tested perfect hashing for subtype testing and method invocation in the PRM compiler and compare it with the coloring technique that amounts to maintaining the single inheritance implementation in multiple inheritance. The results exceed our expectations and confirm that perfect hashing must be considered for implementing JAVA interfaces.

Categories and Subject Descriptors: D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; JAVA; D.3.3 [**Programming languages**]: Language Constructs and Features—*classes and objects*; *inheritance*; D.3.4 [**Programming languages**]: Processors—*run-time environments*; E.2 [**Data**]: Data Storage Representations—*hash-table representations*; *object representations*

General Terms: Experimentation, Languages, Measurement, Performance

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

1. INTRODUCTION

The implementation of object-oriented languages represents an important issue in the context of both *multiple inheritance* and *dynamic loading*. In a recent article [Ducournau 2008], we identified three requirements that all implementations of object-oriented languages, especially in this context, should fulfil—namely (i)

Authors' address: R. Ducournau and F. Morandat, LIRMM, 161, rue Ada – 34392 Montpellier Cedex 5, France

constant-time, (ii) *linear-space* and (iii) *inlining*. This 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 there is no known efficient subtype test 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 [Alpern et al. 2001]). The present research was motivated by this observation—though object-oriented technology is mature, the ever-increasing size of object-oriented class libraries makes the need for scalable implementations urgent and there is still considerable doubt over the scalability of existing implementations.

In the aforementioned article, we proposed a new technique, called *perfect hashing* (PH), for subtyping tests and method invocation. To our knowledge, perfect hashing is the first and only technique that fulfils all five requirements. 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, denoted hereafter `mod`, and bit-wise `and`. However, these two functions involve a tradeoff between space and time efficiency. The former yields more compact tables but the integer division latency may be more than 20 cycles, whereas the latter is a 1-cycle operation but yields larger tables.

In this article, we present the results of recent experiments that show that perfect hashing provides a very efficient implementation for object-oriented languages, in a multiple inheritance and dynamic loading framework.

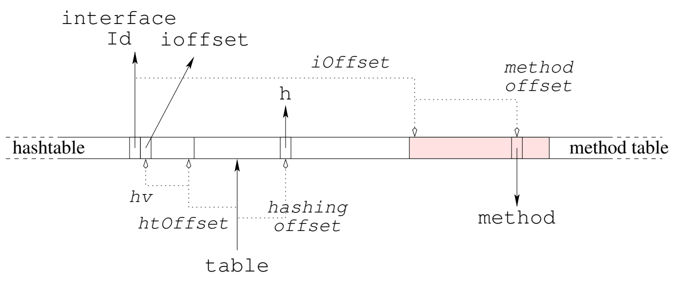
1.1 Perfect Hashing for Object Implementation

The problem 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 $\preceq$ 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. $d \preceq c$. Usually d is the dynamic type of some object and the programmer or compiler wants to check that this object is actually an instance of c. The point is to efficiently implement this test by precomputing some data structure that allows for constant time. Dynamic loading adds a constraint, namely that the technique should be inherently incremental. 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.

The *perfect hashing* principle is as follows. When a class c is loaded, a unique identifier id_c is associated with it and the set $I_c = \{id_d \mid c \preceq d\}$ of the identifiers of all its superclasses is known—if needed, yet unloaded superclasses are recursively loaded. So, $c \preceq d$ iff $id_d \in I_c$. This set I_c is immutable, hence it can be hashed

[1]When this `virtual` keyword is not used, the implementation is markedly more efficient but no longer fully reusable because it yields *repeated inheritance*—so the language is no longer compatible with both multiple inheritance and dynamic loading. In the following, we only consider C++ under the first implementation.

```
                        load [object + #tableOffset], table
                        load [table + #hashingOffset], h
                        and #interfaceId, h, hv
                        sub table, hv, htable
| load [htable +#htOffset-fieldLen], id    | load [htable +#htOffset], ioffset
| comp #interfaceId, id                    | add ioffset, htable, itable
| bne #fail                                | load [itable +#methOffset], method
|                                          | call method
```



Pointers and pointed values are in roman type with solid lines, and offsets are italicized with dotted lines. The grey rectangle denotes the group of methods introduced by the considered interface.

Fig. 1. Perfect Hashing for JAVA—Code sequence in Driesen's pseudo-language, for subtype testing (left) and method invocation (right) and method table layout (bottom)

with some *perfect hashing function* h_c, that is, a hashing function that is injective on I_c [Sprugnoli 1977; Czech et al. 1997]. The previous condition becomes: $c \preceq d$ iff $ht_c[h_c(id_d)] = id_d$, whereby ht_c denotes the hashtable of c. Moreover, the cardinality of I_c is denoted n_c. The technique is obvioulsy incremental since all hashtables are immutable and the computation of ht_c only depends upon I_c. To our knowledge, PH is the only constant-time technique for subtype testing that allows for both multiple inheritance and dynamic loading at reasonable spatial cost.

In a static typing setting, the technique can also be applied to method invocation and we did propose, in the aforementioned article, an application to JAVA interfaces. For this, the hashtable associates, with each implemented interface, the offset of the group of methods that are introduced by the interface. Figure 1 recalls the precise implementation in this context and the corresponding sequence code in the pseudo-language of Driesen [2001]. The method table is bidirectional. Positive offsets involve the method table itself, organized as with single inheritance. Negative offsets consist of the hashtable, which contains, for each implemented interface, the offset of the group of methods introduced by the interface. The object header points at its method table by the `table` pointer. `#hashingOffset` is the position of the hash parameter (`h`) and `#htOffset` is the beginning of the hashtable. At a position `hv` in the hashtable, a two-fold entry is depicted that contains both the implemented interface ID, that must be compared to the target interface ID (`#interfaceId`), and the offset `iOffset` of the group of methods introduced by the interface that introduces the considered method. The table contains, at the position `#methodOffset` determined by the considered method in the method group, the address of the function that must be invoked. 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.

Application to plain multiple inheritance would also need to deal with attribute access. It is possible by associating, with each superclass, the offset of the group of attributes introduced by this class. This makes the hashtable entries 3-fold—the class ID and the offsets of method and attribute groups. Such a technique, called *accessor simulation* in [Ducournau 2006], can be derived from any method invocation technique. However, with perfect hashing, the technique will likely be time-inefficient. Besides subtype testing, which does not depend on whether typing is static or dynamic, the other perfect hashing applications only work in static typing.

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* (noted `mod`) and bit-wise `and`[2]. Both represent a single instruction function. However, only bit-wise `and` is a 1-cycle instruction—actually, the latency of integer division is commonly more than 20 cycles. In both cases, H_c is defined as the least integer such that h_c is injective on the set I_c. The algorithms for computing H_c are both efficient and straightforward (see Appendix B). So the point at issue is memory occupation.

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 this kind of tables should be linear in the cardinality of the specialization relationship $\preceq$. This cardinality is exactly the space required by the technique known as Cohen's display, proposed by Cohen [1991] for single inheritance.

The results of our tests were encouraging but not perfect—namely, PH-`mod` requires a little more than twice the cardinality of $\preceq$, but PH-`and` appears to be much more greedy, especially in the case of a very large benchmark (IBM-SF). So the scalability of PH-`and` was not certain.

These first experiments left a number of issues open:

—First of all, PH parameters depend on the class IDs which are assigned as classes are loaded and numbered, hence they might vary according to class loading orders. However, we only considered a single arbitrary order for loading classes—this ordering was likely a top-down depth-first linear extension that might be far from representative of actual class loading orders.

—We proposed a variant based on *quasi-perfect hashing* (qPH) [Czech 1998], that yields at most one collision per hashed value, hence a 2-probe test; this gives more compact tables at the expense of less efficient code; however, we failed to identify 2-parameter hashing functions that might reduce the table size at the expense of smaller time overhead.

—We proposed to improve perfect hashing by optimizing the identifier id_c of the currently loaded class: however, the technique, called *perfect numbering*, yielded

[2]With `and`, the exact function maps x to $\texttt{and}(x, H_c - 1)$.

strange results, so we did not include them in the article.

—Finally, our assessment of time-efficiency relied on an abstract processor model borrowed from Driesen [2001]; though we believe in the model validity, it should obviously be complemented by empirical time measurement.

1.3 Contributions and Plan

In this paper, we propose answers to these different issues:

—We have extended our testbed in order to generate class loading orders at random; potentially, any order can be generated; though the complete combinatorics is intractable, this gives rough statistics and a strong indication about the behavior of the various techniques.

—We have tested *perfect class numbering* with both `mod` (PN-`mod`) and `and` (PN-and) in this new setting, which better explains the observed odd behavior of PN-`and`. Furthermore, we proved a simple optimality condition for PN-`and` in single inheritance situations.

—We also propose several 2-parameter functions and test them in our extended testbed.

—All of these techniques are then tested based on the assumption that only leaf-classes can have instances—*"make all non-leaf classes abstract"* says Meyers [1996].

—Finally, all of these experiments are repeated with benchmarks that simulate the JAVA distinction between classes and interfaces.

Besides these results, which only address the space-efficiency of perfect hashing, we have implemented PH in the compiler of a real language, PRM [Privat and Ducournau 2005], and tested its time-efficiency on real programs, namely the PRM compiler itself. In this setting, perfect hashing is compared to *coloring*, which is likely the most efficient implementation technique without global optimization [Ducournau 2006], and to an incremental version of coloring proposed by Palacz and Vitek [2003], that we have already compared to perfect hashing in [Ducournau 2008]. The tests have been performed on a variety of processors of the x86 architecture.

Overall, these new tests overcome all of our previous reservations about the use of perfect hashing for implementing object-oriented languages. In a dynamic loading setting, we consider now that the technique is efficient from both space and time standpoints. So it should be considered by language implementors: (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, etc.). However, using perfect hashing for attribute access, hence complete object implementation, requires further research and tests.

The article is structured as follows. Section 2 presents the core of this work, namely hashing functions, experiments and statistical results, in the context of plain multiple inheritance, and then similar experiments and results in the context of JAVA interfaces. All these experiments concern space-efficiency. Section 3 describes the experiments that have been done in the PRM compiler and provide a first empirical assessment of the run-time time-efficiency. Finally, Section 4 discusses conclusions and future works. An appendix gathers some simple mathematical results about bit-wise `and` perfect hashing—especially a lower bound that is proven to be the *perfect numbering* optimal in case of single inheritance—and presents some algorithms for computing PH parameters.

This article is the continuation of [Ducournau 2008]. We have 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). There is no discussion of related works here, since this was done at length in PHAPST and we are not aware of any new results on related topics. Interested readers are referred to PHAPST for a more in-depth presentation and discussion of subtype testing, perfect hashing and all related topics.

2. 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. So this testbed provides simulation of the memory occupation of these various techniques. The simulation is exact and reproducible, except for techniques that rely on heuristics or depend on some run-time input, for instance class loading order. Besides PHAPST, this testbed has been used in different simulations [Ducournau 2009; 2006]. For this article, we have extended this testbed in two directions, by computing random class loading orders and defining new perfect hashing variants.

2.1 Class Loading at Random

In a first set of experiments, we have computed various perfect hashing parameters while loading classes at random. The precise testbed involves generating a random class loading order, by selecting a class at random from the set of *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 I and II present minimum, average and maximum values over randomly generated class loading orders. We now only consider PH-`mod` and PH-`and` columns, whose results presented in PHAPST are also recalled. We have also tested qPH-`and`, but we do not include the results here, as they are not good enough to offset the time overhead. These numbers are all ratios $\rho = \sum_c H_c / \sum_c n_c$ that represent the space-linearity factor. Of course $\rho \geq 1$. Moreover, $\rho = 2$ ensures a good efficiency on average for usual hashtables based on *linear probing* [Morris 1968; Knuth 1973; Vitter and Flajolet 1990]. In Table II, the first column presents a lower bound for bit-wise functions—actually, hashing n numbers requires at least $\lceil \log_2 n \rceil$ 1-bits and, conversely, a mask with k 1-bits can hash a set of 2^k numbers (see Appendix A).

The conclusions of these first experiments are as follows:

—in all cases, the results presented in PHAPST are close to the minimum values, hence rather optimistic;

—for *modulus*, the variations are not really significant—actually, the ratio between the maximum and minimum observed values is never greater than 1.6;

—for bit-wise `and`, the variation can be much more important—the same max/min ratio can be greater than 30; on the other hand, the ratio between the average

Table I. Statistics over random class loading orders with *modulus*

3130	N	n_c	PN-`and+mod`			PN-`mod`			PH-`and+mod`			PH-`mod`			
2.7		avg	min	avg	max	min	avg	max	min	avg	max	ref	min	avg	max
IBM-SF	8793	9.2	1.7	1.9	2.2	1.9	2.2	2.8	2.1	2.2	2.6	2.5	2.5	2.9	3.4
JDK1.3.1	7401	4.4	1.2	1.3	1.4	1.3	1.3	**1.5**	1.5	1.6	1.7	1.9	**1.9**	2.0	2.2
Java1.6	5074	4.4	1.3	1.3	1.4	1.3	1.4	**1.5**	1.5	1.6	1.7	1.9	**1.9**	2.0	2.3
Orbix	2716	2.8	1.1	1.1	1.2	1.1	1.2	**1.3**	1.3	1.4	1.5	1.6	**1.5**	1.7	2.0
Corba	1699	3.9	1.2	1.3	1.5	1.2	1.4	**1.7**	1.5	1.6	1.8	1.7	**1.7**	1.9	2.4
Orbacus	1379	4.5	1.3	1.4	1.6	1.3	1.5	**1.8**	1.6	1.7	1.8	1.8	**1.9**	2.1	2.4
HotJava	736	5.1	1.3	1.5	1.7	1.3	1.5	2.1	1.6	1.7	1.9	1.9	1.8	2.1	2.5
JDK.1.0.2	604	4.6	1.1	1.3	1.5	1.2	1.3	**1.4**	1.5	1.6	1.9	1.8	**1.7**	2.0	2.5
Self	1802	30.9	1.6	2.7	3.3	1.3	*2.4*	*3.2*	1.9	2.2	2.4	2.1	*2.0*	*2.3*	2.5
Geode	1318	14.0	2.1	2.5	3.2	2.2	2.8	*3.8*	2.4	2.7	3.1	3.0	3.0	*3.3*	3.9
Vortex3	1954	7.2	1.5	1.6	1.8	1.6	1.7	**1.9**	1.8	2.0	2.1	2.1	**2.3**	2.5	2.7
Cecil	932	6.5	1.4	1.5	1.7	1.4	1.6	**1.8**	1.7	1.8	2.1	2.0	**2.1**	2.3	2.6
Dylan	925	5.5	1.2	1.3	1.4	1.1	1.2	**1.5**	1.4	1.6	1.9	1.6	**1.7**	1.9	2.4
Harlequin	666	6.7	1.5	1.7	1.8	1.6	1.8	**2.0**	1.8	1.9	2.1	2.0	**2.2**	2.4	2.6
Lov-obj-ed	436	8.5	1.7	1.9	2.1	1.8	2.0	**2.3**	1.9	2.1	2.3	2.3	**2.3**	2.5	2.8
SmartEiffel	397	8.6	1.2	1.4	2.0	1.2	1.3	2.0	1.6	1.8	2.2	1.9	1.7	2.1	2.7
Unidraw	614	4.0	1.1	1.2	1.3	1.1	1.2	**1.3**	1.4	1.5	1.7	1.7	**1.7**	1.8	2.2
PRMcl	479	4.6	1.2	1.3	1.4	1.2	1.3	**1.4**	1.5	1.6	1.8	1.8	**1.7**	1.9	2.2
Total	37925	7.3	1.5	1.9	2.2	1.5	1.9	2.4	1.8	2.0	2.2	2.2	2.2	2.4	2.8

The top-left number is the sample count for each benchmark. The first columns present the class number N of each benchmark and the average value of n_c on all classes. 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 $n_c \geq 1$ is the number of superclasses of c including itself (the denominator is the cardinality of $\preceq$). The minimum, average and maximum values of ρ are presented for each technique and the 'ref' column recalls the tests presented in PHAPST. Line "Total" sums parameter N on all benchmarks and represents, for all other 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 $\sum_b p_b / \sum_b q_b$. Italic numbers are PN avg or max that are greater than the corresponding PH min or avg. Bold numbers are PN max that are less than PH min.

and minimum values is never greater than 2, so most of the class loading orders should yield more acceptable results.

Overall, our previous conclusions about PH-`mod` are confirmed, namely, with this hashing function the ratio ρ is not much greater than 2, hence close to the linear probing optimal. On the contrary, 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. However, all linear extensions are not sensible class loading orders. So an open issue involves modelling class loading orders. This issue will be addressed in Section 2.4. Anyway, the PH-`and` minimum is about twice the PH-`mod` maximum—hence, restricting class loading orders will likely not suffice to make PH-`and` as good as PH-`mod`.

2.2 Perfect Class Numbering

In PHAPST, we proposed a variant of perfect hashing that amounts to optimizing the class identifier id_c in order to minimize the H_c parameter. So, instead of numbering classes as they are loaded, the H_c parameter is first computed from the superclass identifers, before computing the best id_c that fits the resulting hashtable. The approach was called *perfect numbering* (PN). Note that the code generated for PN is exactly the same as that for PH, so they have the same time-efficiency—the only difference involves computation of the H_c and id_c parameters. Moreover, this computation only involves a slight overhead over that of PH (see Appendix B). Hence, PN must always be preferred to PH when the space improvement is effective. We tested *perfect class numbering* in the context of both `mod` and `and` functions.

Table II. Statistics over random class loading orders with bit-wise `and`

3130		PN-`and+shift`			PN-`and`			PH-`and+shift`			PH-`and`			
2.6	$2^{\lceil \log_2(n_c) \rceil}$	min	avg	max	min	avg	max	min	avg	max	ref	min	avg	max
IBM-SF	1.42	2.5	4.5	30.0	3.5	8.6	*76.1*	4.8	8.2	58.6	10.4	10.1	*19.0*	118.8
JDK1.3.1	1.24	1.5	1.8	5.3	1.9	3.5	*32.6*	2.5	3.3	9.9	11.7	6.6	*10.6*	71.2
Java1.6	1.27	1.5	1.9	8.3	2.1	3.7	*32.0*	2.5	3.4	10.8	7.7	6.5	*10.2*	92.6
Orbix	1.13	1.2	1.4	3.2	1.3	1.9	*32.3*	1.6	2.0	23.2	4.4	4.1	*5.9*	67.8
Corba	1.19	1.4	1.9	9.6	1.7	2.9	*34.5*	2.2	3.1	18.6	5.1	4.6	*7.1*	69.7
Orbacus	1.20	1.5	2.3	14.9	2.1	4.0	*34.2*	2.5	3.8	16.5	5.3	5.4	*8.8*	48.2
HotJava	1.31	1.5	2.2	8.0	1.7	3.5	*23.2*	2.5	4.1	15.7	6.4	4.9	*8.0*	27.3
JDK.1.0.2	1.30	1.3	1.5	6.1	1.3	1.8	*11.0*	2.2	3.4	22.5	7.4	4.1	*7.1*	28.3
Self	1.30	2.0	3.9	7.8	2.1	4.0	7.8	6.3	9.2	17.4	5.9	6.3	9.2	17.5
Geode	1.48	2.9	6.7	24.4	4.6	*9.0*	*28.5*	6.0	11.0	26.5	11.5	*8.6*	*15.0*	33.0
Vortex3	1.33	2.0	2.9	7.7	2.6	4.7	*16.3*	3.9	5.9	21.6	11.0	7.2	*12.3*	36.0
Cecil	1.28	1.7	2.5	10.3	1.8	3.8	*12.4*	3.2	5.0	13.3	8.3	5.1	*9.3*	31.8
Dylan	1.35	1.4	1.6	5.1	1.4	1.6	5.1	4.3	6.9	29.2	4.6	4.3	6.9	29.2
Harlequin	1.32	2.0	3.1	7.1	2.3	4.0	*9.1*	3.3	5.3	14.9	5.9	5.8	*9.0*	18.5
Lov-obj-ed	1.38	2.6	4.0	9.3	3.0	5.2	*11.1*	4.0	6.3	11.4	6.3	5.7	*8.7*	16.1
SmartEiffel	1.41	1.4	1.9	12.1	1.4	1.9	*12.1*	4.4	7.0	16.0	4.6	4.4	*7.0*	16.0
Unidraw	1.27	1.3	1.3	2.9	1.3	1.3	**3.0**	1.9	2.7	9.6	4.2	**4.0**	6.1	27.8
PRMcl	1.31	1.3	1.5	6.0	1.4	2.0	*9.0*	2.2	3.1	8.4	4.4	4.0	*6.4*	18.3
Total	1.33	2.0	3.5	15.2	2.6	5.4	*36.5*	4.3	6.8	28.8	8.6	7.3	*12.5*	65.3

The table follows the same convention as Table I. The first column presents the lower bound for bit-wise hashing, as a ratio $\sum_c 2^{\lceil \log_2(n_c) \rceil} / \sum_c n_c$ which is always in interval $[1, 2[$.

Contrary to plain perfect hashing which relies on a straightforward mathematical definition—namely, the H_c parameter is the least integer that makes the hashing function injective—there is not such a simple definition for perfect numbering. It actually consists of a greedy optimization that does not ensure any formal minimization of $\sum_c H_c$. So we only propose a heuristic that might well be improved by further research.

Let $I'_c = \{id_d \mid c \prec d\}$ be the set of identifiers of all of the strict superclasses of c. H_c is now defined as the least integer such that (i) h_c is injective on the set I'_c and (ii) the resulting hashtable has some free entry for an extra identifier. In the `mod` case, H_c is simply defined as the least integer greater than or equal to n_c (the cardinality of I_c) such that h_c is injective on I'_c—hence, the resulting hashtable has at least one empty entry, say i, and for all k, $i + kH_c$ is a suitable identifier for c.

With `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. This follows from the fact that the hashtable capacity depends on the 1-bit count of the mask, not on its magnitude (see Appendix A). So, H_c is now defined as parameter H'_c computed for PH-`and` on I'_c if the bit-wise mask (i.e. $H'_c - 1$) has at least $\log_2 n_c$ 1-bits. Otherwise, when $n_c = 2^k + 1$ and the bit-wise mask $H'_c - 1$ has exactly k 1-bits, the least-weight 0-bit of the mask is switched to 1. This ensures that there is a subset of 1-bits in the mask that forms the offset of an empty entry in the table. Moreover, PN-`and` presents 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 (see Proposition A.5).

With both functions, the identifier id_c is then computed as the least *free* (i.e. not yet used as a class identifier) integer that can be hashed in an empty place in the resulting hashtable.

This technique was tested on all benchmarks with random class loading and the results are presented in Tables I and II. The conclusions are that PN always markedly improves upon PH. More exactly,

—with `mod`, the PN maximum is almost always lower than the PH average (the few exceptions are in italic type in Table I) and often lower than the PH minimum (in bold type);

—with `and`, the PN average is always lower than the PH minimum, except again in a few cases; moreover, the PN minimum is close to PH-`mod`; however, the PN-`and` maximum is generally higher than the PH-`and` average, though markedly lower than its maximum.

Note that PN minimum values are far better than the corresponding PH values because the class numbering associated with PN random are not among those of PH—indeed PH relies on consecutive numbering.

In PHAPST, we did not conclude about PN because our first results seemed erratic. Actually PH-`and` is itself erratic, but PN is always an improvement upon PH. Overall, perfect numbering must always be used instead of perfect hashing. However, the technique remains space-inefficient with bit-wise `and`, at least in the worst-case orders of class loading. But now, the PN-`and` minimum is generally better than the PH-`mod` maximum—hence, restricting class loading orders might make PN-`and` acceptable.

2.3 Two-Parameter Hashing Functions

Two-parameter *hash* functions might be an improvement from the space standpoint, but the extra instruction would likely degrade the time efficiency unless it is 1-cycle. In PHAPST, we actually could not imagine two 1-cyle instructions that would combine efficiently.

2.3.1 *Modulus-Based Functions.* *Modulus* provides space-efficient combinations that are, however, time-inefficient. *Modulus* indeed has interesting properties: (i) it is monotonic in the sense that $\texttt{mod}(x, y) \leq x$ for all positive integers, (ii) it is also monotonic in the sense that $H_c - 1 \leq \max(I_c)$, i.e. the maximum index is lower than the maximum input number, and (iii) it is not associative. Hence, PH-`mod` can be recursively applied to the set $I_c^2 = \{\texttt{mod}(x, H_c) \mid x \in I_c^1\}$ whereby $I_c^1 = I_c$. This yields a series of techniques PH-$\texttt{mod}^i$ and parameters H_c^i that are decreasing and quickly convergent. Moreover, PH-`mod` can also be applied to the set $\{\texttt{and}(x, H_c - 1) \mid x \in I_c\}$—it should improve on PH-`and`.

We tested PH-$\texttt{mod}^2$, PH-$\texttt{mod}^\infty$ and PH-`and+mod`. They are all better than PH-`mod` but only the latter should be considered further because several integer divisions would be time-inefficient. However, the results of PN-`mod` are similar to that of PH-`and+mod`, so PN-`mod` should be preferred. Of course, this combination can also be applied to perfect numbering, and PN-`and+mod` gives slightly better results (Table I).

In contrast, bit-wise `and` does not provide the same monotonicity, because the upper bound of H_C is only $2^{\lceil \log_2(\max(I_c)+1) \rceil} \leq 2\max(I_C)$, so the maximum index is not decreasing and PH-`mod+and` does not always improve on PH-`mod`.

2.3.2 *Bit-Wise Functions.* As bit-wise `and` is associative, it is meaningless to apply it recursively, i.e. PH-$\texttt{and}^\infty$=PH-$\texttt{and}^1$. However, bit-wise `and` can be combined with `shift` in order to truncate the bit-wise mask and remove all trailing

Table III. Statistics over random concrete leaf-class loading orders with *modulus*

25680	class	leaf	n_c	PN-`and+mod`			PN-`mod`			PH-`and+mod`			PH-`mod`		
	number		avg	min	avg	max	min	avg	max	min	avg	max	min	avg	max
IBM-SF	8793	6001	8.8	1.4	1.6	2.0	1.2	1.4	**1.8**	1.7	1.9	2.2	**1.8**	2.0	2.4
JDK1.3.1	7401	5806	4.5	1.2	1.3	1.4	1.1	1.2	**1.3**	1.5	1.5	1.6	**1.7**	1.8	2.0
Java1.6	5074	3825	4.6	1.2	1.3	1.5	1.1	1.2	**1.3**	1.5	1.5	1.7	**1.7**	1.8	2.0
Orbix	2716	2440	2.7	1.0	1.1	1.3	1.0	1.0	**1.2**	1.3	1.3	1.5	**1.4**	1.5	1.8
Corba	1699	1473	3.7	1.1	1.2	1.5	1.0	1.1	1.6	1.4	1.5	1.7	1.5	1.7	2.1
Orbacus	1379	954	4.7	1.1	1.3	1.7	1.0	1.1	**1.4**	1.4	1.6	1.8	**1.6**	1.8	2.2
HotJava	736	525	5.6	1.2	1.4	1.7	1.1	1.3	**1.6**	1.4	1.6	1.9	**1.6**	1.8	2.2
JDK.1.0.2	604	445	4.9	1.1	1.3	1.6	1.0	1.1	**1.4**	1.4	1.5	1.8	**1.5**	1.7	2.1
Self	1802	1134	31.9	1.2	1.4	1.9	1.1	1.2	1.6	1.5	1.8	2.2	1.5	1.6	2.2
Geode	1318	732	14.7	1.4	1.7	2.6	1.3	1.6	*2.5*	1.7	2.0	2.8	1.8	*2.1*	2.9
Vortex3	1954	1216	7.4	1.3	1.5	1.8	1.2	1.4	**1.6**	1.6	1.7	2.0	**1.8**	2.0	2.3
Cecil	932	601	6.8	1.2	1.4	1.7	1.1	1.3	**1.5**	1.5	1.7	2.0	**1.7**	1.9	2.2
Dylan	925	806	5.6	1.1	1.3	1.7	1.0	1.2	*1.9*	1.3	1.5	1.9	1.5	*1.7*	2.4
Harlequin	666	371	7.5	1.3	1.5	1.8	1.2	1.4	**1.6**	1.6	1.8	2.0	**1.7**	2.0	2.2
Lov-obj-ed	436	218	9.9	1.4	1.6	2.0	1.3	1.5	1.9	1.6	1.9	2.2	1.7	2.0	2.5
SmartEiffel	397	311	8.9	1.2	1.3	1.7	1.0	1.1	1.5	1.4	1.6	1.9	1.5	1.7	2.0
Unidraw	614	481	4.0	1.1	1.2	1.4	1.0	1.1	**1.2**	1.3	1.4	1.6	**1.5**	1.7	1.9
PRMcl	479	294	5.1	1.1	1.3	1.5	1.0	1.1	**1.3**	1.4	1.5	1.7	**1.6**	1.7	2.1
Total	37925	27633	7.0	1.3	1.4	1.8	1.1	1.3	**1.6**	1.5	1.7	2.0	**1.7**	1.9	2.2

The first columns present the respective class and leaf numbers and the average value of n_c on all leaves. All other numbers are now ratios $\rho = \sum_c H_c / \sum_c n_c$ where the sum is restricted to leaf-classes.

zeros. In that case, the PH-`and` parameter is defined in a slightly different way. H_c is no longer the least integer that makes h_c injective and an extra constraint is required to minimize the distance between the highest and lowest 1-bits (see the algorithm in Appendix B.4). Moreover, `and` and `shift` can also be combined with perfect numbering. We tested PH-`and+shift` and PN-`and+shift`—the results are presented in Table II. The effect of this combination of `and` and `shift` is similar to that of perfect numbering, i.e. the statistics are roughly twofold lower. The combination with perfect numbering is still better—on average, the result is close to PH-`mod` but, in the worst-case orders, the resulting size remains more than twice that of the PH-`and` estimation in PHAPST. So PN-`and+shift` might be the solution if we can be sure that the worst-case orders are excluded.

2.4 Leaf-Class Loading at Random

The main issue with these random tests is that bit-wise `and` is highly dependent on the class loading order but it is likely that all linear extensions are not sensible class loading orders. Class loading depends on precise implementation of runtime systems like virtual machines. However, without loss of generality, one may assume that class loading is always triggered by the need to instantiate a yet unloaded class—all other uses of yet unloaded classes could be made lazy. Hence, only *concrete*, i.e. nonabstract, classes must be ordered and *abstract classes* are only inserted when needed in concrete class orders. As our benchmarks do not record the fact that a class is or is not abstract (See Appendix B in PHAPST for a discussion of these benchmarks), we consider an assumption that is often advocated—*"make all non-leaf classes abstract"* [Meyers 1996]. This is indeed a common methodological advice—see for instance [Steimann 2000]. Table III presents the leaf number of all benchmarks. On average, according to this assumption, there would be a little more than one abstract class to four classes. This is obviously 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 [1994] (however,

Table IV. Statistics over random leaf-class loading orders with bit-wise **and**

19025	$2^{\lceil \log_2(n_c) \rceil}$	PN-and+shift			PN-and			PH-and+shift			PH-and			
		min	avg	max	min	avg	max	min	avg	max	ref	min	avg	max
IBM-SF	1.42	2.2	3.0	7.5	2.2	3.1	**6.2**	6.7	8.8	15.1	10.4	**7.1**	9.3	40.0
JDK1.3.1	1.24	1.4	1.6	5.0	1.6	2.0	6.6	3.9	5.6	20.2	11.7	5.7	8.1	34.4
Java1.6	1.27	1.4	1.7	4.2	1.6	2.1	6.2	3.8	5.7	13.1	7.7	5.6	7.7	20.2
Orbix	1.13	1.1	1.2	6.0	1.2	1.3	3.8	1.7	2.5	7.5	4.4	3.8	4.7	30.1
Corba	1.19	1.3	1.8	5.9	1.4	2.0	*6.1*	2.6	3.9	10.8	5.1	4.0	*5.4*	25.7
Orbacus	1.20	1.3	1.6	8.0	1.4	1.9	*9.6*	3.3	5.0	14.3	5.3	4.1	*6.1*	18.4
HotJava	1.31	1.4	1.8	3.8	1.5	2.0	**3.9**	3.1	4.7	7.7	6.4	**3.9**	5.6	9.8
JDK.1.0.2	1.30	1.3	1.5	3.6	1.3	1.6	3.8	2.7	4.4	9.7	7.4	3.6	5.3	10.7
Self	1.30	1.4	1.6	2.4	1.3	1.6	**2.4**	4.8	5.8	10.5	5.9	**4.8**	5.8	10.5
Geode	1.48	2.1	3.0	7.9	2.2	3.2	*8.5*	5.2	7.3	13.7	11.5	5.3	*7.5*	13.8
Vortex3	1.33	1.7	2.4	4.9	1.8	2.5	**5.0**	4.9	7.3	11.8	11.0	**5.4**	7.9	13.0
Cecil	1.28	1.4	1.9	4.3	1.5	2.1	4.8	4.2	6.0	12.1	8.3	4.6	6.6	12.4
Dylan	1.35	1.4	1.6	12.1	1.4	1.6	*12.1*	3.9	5.5	20.6	4.6	3.9	*5.5*	20.6
Harlequin	1.32	1.7	2.4	4.2	1.8	2.5	5.1	3.9	5.8	9.6	5.9	4.2	6.3	10.6
Lov-obj-ed	1.38	1.8	2.6	4.7	1.8	2.7	4.9	3.9	5.5	8.4	6.3	4.1	5.8	8.9
SmartEiffel	1.41	1.4	1.5	2.7	1.4	1.5	**2.7**	3.8	4.9	7.1	4.6	**3.8**	4.9	7.1
Unidraw	1.27	1.3	1.4	2.0	1.3	1.4	**2.0**	2.4	3.6	6.3	4.2	**3.3**	4.8	10.2
PRMcl	1.31	1.4	1.5	3.5	1.4	1.7	3.7	2.6	4.1	9.7	4.4	3.4	5.0	10.8
Total	1.33	1.7	2.2	5.5	1.7	2.3	**5.5**	4.9	6.6	13.5	8.6	**5.5**	7.4	24.2

The table presents exactly the same data as Table II, except that the statistics are restricted to the subset of class loading orders that are triggered by leaves. Italic numbers are PN max that are greater than the corresponding PH avg. Bold numbers represent either PN minimums that are optimal, or PN maximums that are less than the corresponding PH minimums.

at a time when class hierarchies were markedly smaller).

The informal algorithm is as follows. A 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 ordered in a top-down linear extension. For each class x in this order, id_x and H_x are computed. In the PN case, two algorithms can be considered. The first one only substitutes PN for PH in the previous algorithm—each class in X''_c is considered separately. An alternative is a global optimisation of X''_c numbering—the H_c parameter is computed from the set $I'_c = \{id_x \mid x \in X'_c\}$, in such a way that the resulting table has enough free places for the elements in X''_c. There are actually pros and cons for both approaches, since PN-and is optimal for single inheritance classes (Proposition A.5). So mixing both strategies is certainly better. However, our experiments show that global optimization gives slightly better results. Anyway, with both hashing functions this involves straightforward extensions of the perfect numbering functions (see Appendix B for more details).

There is however a huge number of linear extensions—namely, in this restricted setting, factorial of the number of leaves which may be as many as 6000 in the largest benchmarks (Table III). So the number of orders is above 10^{20000}—obviously, the point cannot be to compute exact statistics. It is possible to somewhat reduce this combinatorial explosion because the set of leaves can be partioned according to their parents—the way leaves with the same parents are ordered with each other is not significant. Therefore, a more efficient algorithm involves taking a set of equivalent leaves at random, then the first element in this set. In this way, the combinatorics reduces to about 10^{12000} orders[3]—this is better though yet exhaustively intractable.

[3]The exact number of orders is now $\text{fact}(\sum_{i=1}^{k} p_i) / \prod_{i=1}^{k} \text{fact}(p_i)$, whereby k is the number of equivalence classes and p_i the cardinality of each of them.

Table V. Statistics over random concrete leaf-class loading orders with bit-wise `and`

25680	$2^{\lceil \log_2(n_c) \rceil}$	PN-`and+shift`			PN-`and`			PH-`and+shift`			PH-`and`		
		min	avg	max	min	avg	max	min	avg	max	min	avg	max
IBM-SF	1.43	2.0	3.0	9.2	2.1	3.1	8.2	6.8	9.2	19.0	7.4	9.9	55.8
JDK1.3.1	1.24	1.3	1.6	5.2	1.4	1.9	7.9	3.9	5.8	23.0	5.9	8.3	41.1
Java1.6	1.28	1.4	1.7	4.5	1.5	2.0	6.8	4.0	6.0	14.5	5.8	8.1	18.3
Orbix	1.11	1.1	1.2	6.6	1.1	1.3	*6.6*	1.7	2.3	7.9	3.8	*4.8*	32.3
Corba	1.18	1.2	1.7	6.4	1.2	1.9	*7.1*	2.5	3.7	11.8	4.0	*5.5*	28.6
Orbacus	1.20	1.2	1.5	10.2	1.2	1.8	*10.7*	3.4	5.3	18.3	4.5	*6.5*	26.4
HotJava	1.33	1.4	1.8	3.7	1.5	2.0	4.2	3.3	5.1	8.5	4.1	6.0	11.0
JDK.1.0.2	1.32	1.3	1.4	3.9	1.3	1.5	4.8	2.8	4.7	9.4	3.7	5.7	11.2
Self	1.33	1.4	1.5	2.8	1.3	1.5	**2.8**	4.9	6.1	12.8	**4.9**	6.1	12.8
Geode	1.48	1.9	2.8	11.5	1.9	2.9	*11.7*	5.3	7.9	20.1	5.4	*8.1*	20.2
Vortex3	1.33	1.5	2.2	5.4	1.6	2.4	**5.5**	5.2	7.7	14.4	**5.8**	8.3	15.3
Cecil	1.27	1.4	1.8	3.9	1.4	2.0	5.1	4.3	6.4	10.8	4.7	7.0	12.5
Dylan	1.37	1.4	1.6	13.6	1.4	1.6	*13.6*	3.9	5.7	22.4	3.9	*5.7*	22.4
Harlequin	1.33	1.5	2.2	5.0	1.6	2.2	4.8	4.1	6.4	10.9	4.6	6.8	11.5
Lov-obj-ed	1.41	1.7	2.5	4.4	1.8	2.6	4.8	4.0	5.9	9.0	4.2	6.1	9.2
SmartEiffel	1.42	1.4	1.5	2.3	1.4	1.5	**2.3**	4.1	5.2	8.3	**4.1**	5.2	8.3
Unidraw	1.27	1.3	1.3	1.9	1.3	1.3	**2.0**	2.4	3.7	6.4	**3.6**	4.9	10.4
PRMcl	1.32	1.3	1.4	3.5	1.3	1.6	3.7	2.8	4.5	11.4	3.5	5.4	12.1
Total	1.33	1.6	2.1	6.4	1.6	2.2	6.8	4.9	6.8	16.3	5.7	7.8	30.8

Like in Table III, all sums are now restricted to leaf-classes.

Though 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.

So, we tested all perfect hashing functions on the same set of benchmarks, under this new assumption. We did it under two forms:

—Table IV presents the same statistics as in Table II except that they are restricted to the subset of class loading orders triggered by leaf-classes. However, in the ratio $\sum_c H_c / \sum_c n_c$, sums are still applied to all classes. The analogue for `mod` is not presented because it does not sufficiently differ from the second form.

—Tables III and V present the same statistics when sums in $\sum_c H_c / \sum_c n_c$ are now restricted to leaf classes—indeed only concrete classes require a method table.

In this setting, the results of both function families are markedly better. For instance, PN-`mod` often results in almost *minimal perfect hashing*—actually, in the best cases, even though the ratio ρ is close to 1, all h_c are not minimal perfect hashing functions, i.e. functions such that $H_c = n_c$ (Table III). Furthermore, modulus does not present the same optimality property as bit-wise `and` with Proposition A.5. So, with `mod`, it would seem that $H_c = n_c$ is a matter of chance. The PH-`and` minimum values are also markedly improved (Tables II and IV). As the sums run on the same set of classes, this only confirms that our random testbed cannot explore all possible orders, 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. On average, the resulting H_c parameter does not significantly differ, whether all yet unloaded classes are numbered as a whole, or whether they are numbered one by one by successive applications of PN in a top-down ordering. Hence, the gain is likely mostly due to the selection of specific class loading orders. In contrast, the comparison between Tables IV and V does not show significant differences according to whether the sums are done over all

Table VI. Statistics for JAVA over random class loading orders with *modulus*

15975	class	intf	n_c	PN-`and+mod`			PN-`mod`			PH-`and+mod`			PH-`mod`			
	number		avg	min	avg	max	min	avg	max	min	avg	max	ref	min	avg	max
IBM-SF	7920	873	6.2	1.4	1.7	2.0	1.5	*1.7*	*2.1*	1.6	1.8	2.3	2.3	*1.7*	*1.9*	2.4
JDK1.3.1	7056	345	1.2	1.6	1.7	1.9	1.6	1.8	*2.0*	1.7	1.8	2.0	2.0	1.8	*2.0*	2.3
Java1.6	4517	1057	1.7	1.3	1.4	1.5	1.3	1.5	*1.7*	1.4	1.5	1.7	1.6	1.5	*1.7*	1.9
yJava1.6	4876	1057	1.6	1.3	1.4	1.6	1.4	1.5	1.7	1.4	1.5	1.7	1.7	1.6	1.7	2.0
xJava1.6	4804	270	1.5	1.5	1.6	1.7	1.5	*1.6*	*1.8*	1.5	1.7	1.9	1.7	*1.6*	*1.8*	2.0
Orbacus	1297	82	1.7	1.4	1.6	1.9	1.5	*1.7*	*2.2*	1.5	1.8	2.1	1.9	*1.6*	*1.9*	2.4
Corba	1634	65	1.1	1.7	1.9	2.1	1.8	*2.0*	*2.3*	1.8	2.0	2.2	2.0	*1.8*	*2.1*	2.5
HotJava	681	55	2.4	1.3	1.4	1.9	1.3	*1.4*	*1.7*	1.3	1.5	1.9	1.8	*1.4*	*1.6*	2.1
Orbix	2676	40	0.3	3.4	3.5	3.8	3.4	*3.5*	*3.8*	3.4	3.6	4.0	3.7	*3.5*	*3.6*	4.0
JDK.1.0.2	576	28	1.1	1.4	1.6	2.0	1.4	*1.6*	*2.3*	1.4	1.6	2.1	1.7	*1.5*	*1.7*	2.3
Total	36037	3872	2.4	1.5	1.6	1.9	1.5	*1.7*	*2.0*	1.6	1.8	2.1	2.1	*1.7*	*1.9*	2.3

The first columns represent the respective numbers of classes and interfaces, and the average n_c value which is now the number of interfaces implemented by the class c. All other numbers are ratios $\rho = \sum_c H_c / \sum_c n_c$, whereby the sum is done on classes only and H_c is the hashtable size and $n_c \geq 0$ the number of interfaces implemented by c (the denominator is the cardinality of the `implements` relationship).

classes or only concrete leaf-classes.

The most interesting observation is that the erratic behavior of PH-`and` has almost disappeared and the comparison between Tables II and IV (or V) shows that ordering leaf-classes rules out most worst-case orders. Very few ratios ρ exceed 10 for PN-`and` and they mostly concern benchmarks with a low n_c average. Hence they should not yield overconsuming tables. Only Geode combines high n_c average with a worst-case ratio that exceeds 10—however, the average ratio is far lower, about 3. Overall, PN-`and` produces excellent results. It is often very close to its lower-bound $2^{\lceil \log_2(n_c) \rceil}$, though it does not exclude a risk of a bad-case class loading order. However this risk is low, as the bad cases are unfrequent, and 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—PN-`and+shift` is not sufficiently better to counterbalance its expected time-overhead.

Of course, it would be interesting to confirm that a leaf-class order is a good model of any class loading order. The reality is likely midway between both models. Programmers and class hierarchies only partly comply with Meyers' directive. The worst-case orders might not be sensible class loading orders, but real class loading orders might have PH parameters that are not as perfect as those of leaf-class orders.

2.5 Application to JAVA

In PHAPST, perfect hashing was also applied to JAVA interfaces in a restricted form close to the restriction to concrete classes. In JAVA, only the `implements` relationship between classes and interfaces needs to be hashed—the `extends` relationship between classes is indeed in single inheritance, so Cohen's display and usual single inheritance implementation apply more efficiently. Overall, only classes require a hashtable and only interfaces are numbered and hashed—this has an important consequence that n_c can be zero whereas H_c must be at least one.

We have adapted all perfect hashing and perfect numbering functions to JAVA interfaces and tested them on the same benchmarks as in previous work. The algorithm is exactly the same as that for leaf-class ordering, except that c is randomly selected among maximal unloaded classes and X'_c and X''_c are the sets of, respec-

Table VII. Statistics for JAVA over random class loading orders with bit-wise `and`

15975	$2^{\lceil \log_2(n_c) \rceil}$	PN-`and+shift`			PN-`and`			PH-`and+shift`			PH-`and`			
		min	avg	max	min	avg	max	min	avg	max	ref	min	avg	max
IBM-SF	1.42	2.3	3.3	5.8	2.3	3.3	5.8	3.9	5.9	10.6	6.6	4.0	5.9	10.6
JDK1.3.1	1.65	1.8	2.3	4.2	2.0	2.7	*5.3*	2.4	3.4	6.7	3.2	2.7	*4.0*	7.5
Java1.6	1.35	1.5	2.0	9.7	1.6	*2.7*	*17.1*	1.9	3.2	19.3	3.5	*2.4*	*4.9*	24.2
yJava1.6	1.39	1.5	2.0	9.5	1.7	*2.7*	*16.5*	1.9	3.2	16.9	3.0	*2.5*	*5.0*	20.5
xJava1.6	1.50	1.7	2.1	3.7	1.8	2.5	*5.3*	2.1	3.0	4.7	3.2	2.6	*3.6*	6.5
Orbacus	1.58	1.6	2.2	4.8	1.7	*2.4*	*5.7*	2.0	3.3	7.3	2.4	*2.1*	*3.7*	7.5
Corba	1.81	2.0	2.5	4.1	2.2	*2.6*	*4.2*	2.2	3.1	5.1	2.6	*2.4*	*3.2*	5.4
HotJava	1.44	1.5	1.8	3.1	1.6	*2.0*	*3.4*	1.7	2.3	3.8	2.5	*1.8*	*2.5*	4.1
Orbix	3.54	3.5	3.7	5.1	3.5	*3.7*	*5.1*	3.6	4.0	5.2	4.0	*3.6*	*4.1*	5.5
JDK.1.0.2	1.67	1.7	1.8	4.0	1.7	*1.9*	*4.0*	1.7	1.9	4.5	1.8	*1.7*	*2.1*	4.5
Total	1.48	2.0	2.8	6.0	2.1	3.0	*7.5*	3.1	4.7	10.6	5.0	3.3	*5.2*	11.6

As n_c can be zero, the first column represents the optimal ratio $\rho = \sum_c o_c / \sum_c n_c$, where $o_c = 1$ if $n_c = 0$ and $2^{\lceil \log_2(n_c) \rceil}$ otherwise. This explains why the ratio can be greater than 2 in the Orbix case.

Table VIII. Statistics for JAVA over random leaf-class loading orders with bit-wise `and`

30545	$2^{\lceil \log_2(n_c) \rceil}$	PN-`and+shift`			PN-`and`			PH-`and+shift`			PH-`and`		
		min	avg	max	min	avg	max	min	avg	max	min	avg	max
IBM-SF	1.44	1.8	2.6	5.0	1.9	2.7	4.7	3.6	5.2	8.8	3.7	5.2	8.9
JDK1.3.1	1.66	1.8	2.2	5.5	1.9	2.5	*5.8*	2.3	3.3	7.7	2.6	*3.9*	10.4
Java1.6	1.35	1.5	1.7	5.0	1.6	2.1	*8.5*	1.9	2.9	9.3	2.4	*4.0*	11.2
yJava1.6	1.35	1.4	1.7	5.3	1.6	2.1	*8.9*	2.0	2.9	9.7	2.5	*4.0*	11.7
xJava1.6	1.49	1.7	2.0	4.2	1.8	*2.4*	*5.3*	2.0	2.9	5.9	*2.3*	*3.6*	9.2
Orbacus	1.58	1.6	1.9	5.6	1.6	*2.1*	*5.9*	2.0	3.0	7.0	*2.0*	*3.3*	8.7
Corba	1.96	2.0	2.4	2.9	2.1	*2.4*	*4.0*	2.3	2.8	4.0	*2.3*	*2.9*	6.0
HotJava	1.42	1.5	1.9	3.5	1.5	*2.0*	*3.7*	1.7	2.4	4.4	*1.8*	*2.7*	4.6
Orbix	4.24	4.2	4.3	4.7	4.2	*4.3*	*5.5*	4.2	4.5	5.5	*4.3*	*4.6*	6.5
JDK.1.0.2	1.66	1.7	1.7	4.1	1.7	*1.7*	*4.1*	1.7	1.8	4.6	*1.7*	*1.9*	4.7
Total	1.50	1.8	2.3	4.9	1.8	2.5	*5.6*	2.9	4.1	8.3	3.1	*4.5*	9.3

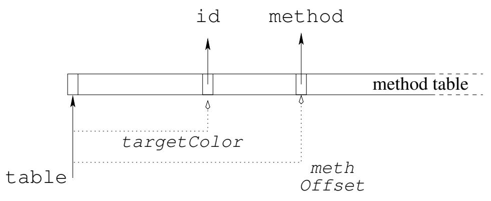
n_c represents now the average on all leaves of the number of implemented interfaces.

tively, already loaded and still unloaded interfaces implemented by c. Most of our benchmarks do not record the difference between classes and interfaces, so we computed this difference according to simple heuristics. See Appendix B in PHAPST for a discussion on the validity of these heuristics—the Java1.6 benchmark has been generated in order to provide an empirical assessment of these heuristics and its different variants show similar behaviors though the interface numbers differ.

Tables VI and VII present the statistics over random class loading orders and Table VIII presents the same statistics when only leaf-classes are ordered. The analogue is not presented for `mod` because it does not sufficently differ from Table VI. One observes that the results are very similar to those of the corresponding Tables I to V—the main difference here is that the linear-space criterion is relative to the cardinality of the `implements` relationship. As a class can implement zero interface, n_c can be 0 while H_c is at least 1. Hence, the ratio could be infinite, when a class hierarchy does not implement any interface. Of course, it is unrealistic but it explains why the ratio ρ can be much higher than in multiple inheritance tests—e.g. for the Orbix benchmark which has very few interfaces. Another difference is that PH-`and` is markedly less erratic than with plain multiple inheritance. Moreover, the statistics with all-class and leaf-class orders do not markedly differ. This is likely due to the fact that ordering only classes implies a structure that is strong enough to avoid most worst-case orders.

Anyway, the overall conclusion is that PN-and would also give excellent results for JAVA interfaces.

```
                      load [object + #tableOffset], table
load [table + #targetColor], id            load [table +#methOffset], method
comp #targetId, id                         call method
bne #fail
```



The figure follows the same convention as Figure 1. When coloring is applied on the whole class hierarchy, method invocation and subtype testing involve only direct access in the method table (pointed by `table`), as in single inheritance.

Fig. 2. Coloring for full multiple inheritance

3. TIME-EFFICIENCY TESTS

Our abstract assessment in PHAPST of the run-time efficiency of perfect hashing should obviously be confirmed by actual run-time experiments. This section presents these experiments and the results.

3.1 The PRM testbed

These experiments are original, as they compare different implementation techniques, in a common framework that allows a fair comparison, *all other things being equal.*

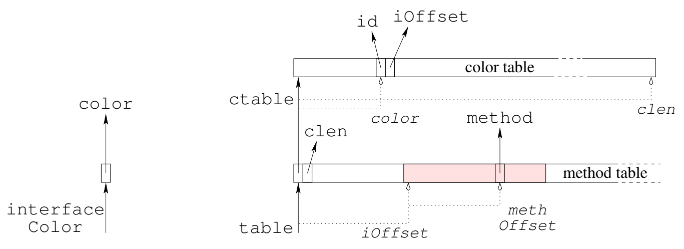
Tested techniques. Three families of techniques are compared:

—The best known constant-time technique, namely *coloring*, generalizes to multiple inheritance the usual single inheritance implementation, at the expense of a global computation [Ducournau 2006]. Coloring represents, for instance, the technique used in JAVA for implementing classes, hence for method invocation and subtype testing when the receiver's type or target type is a class (Figure 2), and for all attribute accesses. With multiple inheritance, coloring requires a global optimization which has no impact on the run-time time efficiency.

—In PHAPST, we also compared perfect hashing with *incremental coloring* (IC), that has been proposed for subtype testing by Palacz and Vitek [2003] in a JAVA real-time setting. As coloring is not inherently incremental, its use with both dynamic loading and multiple inheritance yields marked overhead at load-time, since some recomputation can be required, and at run-time, since these possible recomputations increase the number of memory accesses, while degrading their locality (Figure 3). Overall, our abstract estimation was that incremental coloring should not be better than perfect hashing, or at least not sufficiently better to counterbalance the load-time overhead.

—The most promising variants of perfect hashing have been tested, namely PH-`mod`, PH-`and` and PH-`and+shift`. As the tests concern only time-efficiency, there is no difference between PH and PN.

```
                  load [object + #tableOffset], table
                  load [table + #ctableOffset], ctable
                  load [#interfaceColor], color
                  add ctable, color, ctable
| load [table + #ctableOffset+4], clen     | load [ctable +#fieldLen], ioffset
| comp color, clen                         | add ioffset, table, itable
| bgt #fail                                | load [itable +#methOffset], method
| load [ctable], id                        | call method
| comp #interfaceId, id                    |
| bne #fail                                |
```



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).

Fig. 3. Incremental coloring of JAVA interfaces

Moreover, 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. Perfect hashing was originally proposed for subtype testing. However, in our PRM testbed, there are very few runtime subtype checks compared to, say JAVA 1.4 programs, so a comparison based only on subtype testing would not have been significant. Fortunately, PH also applies to method invocation which makes the comparison much more significant. For PH and IC, the technique for method invocation follows the scheme proposed in PHAPST for JAVA interfaces (Figure 1, page 3)—that is, each table entry contains the class or interface ID for subtype test and the offset of the group of methods introduced by the class or interface, for method invocation. For attribute access, two variants are considered. Attribute coloring is the most efficient implementation, that of JAVA for instance, so combining PH or IC with attribute coloring provides a sound assessment of the considered techniques when applied to JAVA interfaces. *Accessor simulation* is a general alternative for attribute access that relies on any method invocation technique [Ducournau 2006]. Attributes are grouped by introduction class and the group offset is placed in the method table, as if it were a method. Here it is associated with the class ID in the method table (for coloring), in the hashtable (for PH) or in the color table (for IC). So the tests with accessor simulation provide an assessment of the use of the considered technique (IC or PH) for implementing full multiple inheritance. Overall, 5 techniques for method invocation and subtype testing have been compared, each one with 2 attribute access techniques.

Tested program. So we implemented all these techniques in the PRM compiler, which is dedicated to exhaustive assessment of various implementation techniques

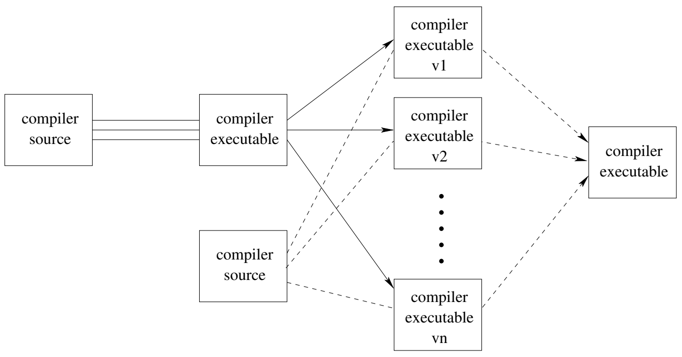


Fig. 4. The PRM testbed
Some compiler source is compiled by some compiler executable, according to different options, thus producing different variants v_1, .., v_n, of the same executable compiler (solid lines). Another compiler source (possibly the same) is then compiled by each variant, with all producing exactly the same executable (dashed lines), and the compilation time is measured.

Table IX. Static and dynamic characteristics of the PRM compiler

number of	static	dynamic
classes	511	
method introductions	2525	
method definitions	4269	
attributes	614	
method calls	14530	2600 M
attribute accesses	4323	300 M

The "static" column depicts the number of program elements (classes, methods and attributes) and the number of sites for each mechanism. The "dynamic" column presents the number of mechanism invocations at run-time (in millions).

and compilation schemes [Privat and Ducournau 2005; Morandat et al. 2009]. The benchmark program is the compiler itself, which is written in PRM and compiles the PRM source code into C code. There are a lot of compilers in the picture, so Figure 4 depicts the precise testbed.

In these tests, the C code generated by the PRM compiler is either the code for global or incremental coloring, or the code for PH. The PRM compiler is actually not compatible with dynamic loading but the code for PH or IC has been generated exactly as if it were generated at load-time. The usual optimizations of the PRM compiler (see [Privat and Ducournau 2005]) are deactivated in all cases. The class load ordering does not matter since we here only consider time measurement. Although these tests represent a kind of simulation, they must be quite reliable. Only the effect of cache misses is likely underestimated, especially for incremental coloring—all color tables are here allocated in the same memory area, whereas load-time recomputations should scatter them in the heap. The only thing that is not considered at all, here, is load-time computation, but our previous analysis shows that it is not significant for perfect hashing in both practice and theory (see Appendix A in PHAPST).

Table IX presents the static characteristics of the PRM compiler, namely the number of different entities that are counted at compile-time, together with the run-time invocation count for each mechanism. These statistics show that the program

Table X. Compilation time according to implementation techniques and processors

processeur	1	Xeon Prestonia		2	Xeon Irwindale		3	Core T2400	
frequency		1.8 GHz			2.8 GHz			2.8 GHz	
cache		512 K			2048 K			2048 K	
reference time	108.7s	2001		53.2s	2006		44.0s	2008	
technique	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
Coloring	0	5.4	5.4	0	9.8	9.8	0	13.6	13.6
IC	3.8	14.1	10.0	5.3	23.3	17.1	5.8	33.0	25.8
PH-`and`	4.9	19.4	13.8	4.3	30.0	24.6	8.5	46.8	34.9
PH-`and+shift`	4.6	20.7	15.4	14.1	51.9	33.1	7.7	47.8	37.2
PH-`mod`	46.8	194.8	100.8	70.4	245.9	103.0	30.9	147.3	88.9

Each subtable presents the results for a precise processor, with the processor characteristics and the reference compilation time. All other numbers are percentage. Each row describes a method invocation and subtype testing technique. The first two columns represent the overhead vs pure coloring, respectively with attribute coloring (AC) and accessor simulation (AS). The third column is the overhead of accessor simulation vs attribute coloring.

size is significant and that it makes intensive usage of object-oriented features.

Time-measurement conditions. Tested variants differ only by the underlying implementation technique, with all other things being equal. Moreover, this is true when considering executable files, not only the program logic. Indeed, the compilation testbed is deterministic—that is, two compilations of the same program by the same compiler executable produce exactly the same executable. This means that: (i) the compiler always proceeds along the program text and the underlying object model in the same order; (ii) the memory locations of program fragments, method tables and objects in the heap are roughly the same. So two compiler variants differ only by the code sequences of the considered techniques, all program components occurring in the executables in the same order. Moreover, when applied to some program, two compiler variants v_i and v_j produce exactly the same code. All program equalities have been verified with the `diff` command on both C and binary files. Overall, the effect of memory locality should be roughly constant, apart from the specific effects due to the considered techniques[4].

The tests were performed on several processors, from the Intel® Pentium^{TM} or AMD® Athlon^{TM} families, all under Linux Ubuntu 8.4, with `gcc` 4.2.4. Two runs of the same compiler on the same computer should take the same time were it not for the noise produced by the operating system. A solution involves running the tests under Linux recovery-mode boot. It has been done for some processors (e.g. 1, 2, 5) but this was actually not possible for remote computers. Finally, a last impediment concerned laptops. Modern laptop processors (e.g. processor 5) are frequency-variable. The frequency is low when the processor is idle or hot. When running a test, the processor must first warm up before reaching its peak speed, then it finishes by slowing down for cooling down. So the peak speed can be very high but only on a short duration. Inserting a pause between each two runs seemed to fix the point. Overall, considering that the difference between two runs of the same executable is pure noise, we took, for each measure, the minimum value of 10

[4]In early tests, compilation was not deterministic and the variation of compilation times between several generations of the same compiler was marked. Hence, the variation between different variants was both marked and meaningless.

Table XI. Compilation time according to implementation techniques and processors (cont.)

processeur	4	Athlon 64		5	Core2 T7200		6	Core2 E8500	
frequency		2.2 GHz			2.0 GHz			3.16 GHz	
cache	43.4s	1024 K		36.0s	4096 K		22.2s	6144 K	
reference time		2003			2006			2008	
technique	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
Coloring	0	21.9	21.9	0	16.6	16.6	0	14.1	14.1
IC	14.6	53.4	33.9	6.8	35.2	26.6	6.5	34.1	25.9
PH-`and`	18.1	73.1	46.6	7.3	45.7	35.7	7.6	45.2	35.0
PH-`and+shift`	18.2	73.5	46.8	7.5	46.0	35.8	7.5	45.3	35.1
PH-`mod`	110.2	345.8	112.1	23.9	158.6	108.6	25.0	119.8	75.8

runs or more.

3.2 Results and Discussion

Tables X and XI presents, for all variants and processors, the time measurement and overhead with respect to the full coloring implementation. Overall, notwithstanding some exceptions that will be discussed hereafter, these tests show that:

—when used for method invocation and subtype testing, PH-`and` yields very low overhead of about 5-7%—this is better than expected;

—the extra instructions of PH-`and+shift` entails low extra overhead when used only for methods and subtyping; on most processors, the difference from PH-`and` is actually far below the precision of measurement;

—when also used for attribute access, the overhead of PH-`and` is much higher, though still reasonable (between 20 and 30%) on some processors, but less reasonable on others;

—incremental coloring (IC) is close to PH-`and`, without significant difference;

—in contrast, the overhead of PH-`mod` is much higher and highly variable, between 24 and 100%, when only used for method invocation and subtype tests;

—finally, when used for attribute access, PH-`mod` becomes unreasonable as it always doubles or triples the compilation time.

The processor influence is also significant, even though it does not reverse the conclusions. Most of them present similar behaviour, although several provide some specific exceptions that make them unique—double overheads on processor 4, rather efficient PH-`mod` on proc. 5 and 6, marked overhead for PH-`and+shift` on proc. 2. These aberrations might be explained, either by some artefact in the experiment, or by some specific feature of the processor—for instance, processor 4 is both an early 64-bit that might be specially inefficient and a remote computer that might be specially noisy. Processors are presented and numbered in the decreasing order of the reference duration, which is strongly correlated with the manufacturing time. It is however hard to find strong correlations between the observed overheads and time or overall performance.

These empirical results can be partly explained by comparing them to the theoretical predictions done in PHAPST according to the computational model proposed by [Driesen 2001]. Table XII sums up this abstract time-efficiency analysis. The Driesen computational model accounts for pipe-line architectures and instruction-level parallelism, though the impact of the latter within PH code sequences concerns

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

	method call			subtype test			attribute access		
technique	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 count and code length that are presented in PHAPST, according to the computational model of [Driesen 2001] which is illustrated in Figures 1, 2 and 3. L, B and D represent the respective latencies of memory loads, undirect or mispredicted branches and integer division. The considered values 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.

only PH-`and+shift` and IC. However, there may be much more parallelism between these code sequences and the rest of the code—this is certainly the case for attribute access, since the single coloring load can be easily parallelized.

—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; the few 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 without significant overhead.

—PH-`mod` adds high integer division latency, about 20-25 cycles, that is much more expensive than extra `loads`—hence marked overhead; according to this latency and the number of method invocations (Table IX), the overhead should be about 20 seconds on all processors; however, the observed overhead runs from 5 seconds (proc. 6) to 40 (proc. 4)—the fact that calls to FPU break the pipeline on Pentium architecture likely accounts for the difference for the latter; however, it does not explain the performance of processors 5 and 6.

—For attribute access, accessor simulation 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.

—In Table XII, the difference between IC and PH-`and` is not significant, and the tests confirm it—extra indirections in IC and extra computations in PH-`and` cause similar overheads. However, as IC implies access to two new memory areas, a first one for the color and a second one for the recomputable color table, extra cache misses are expected, that are likely underestimated in our tests. As the PH load-time cost is also far lower than IC, PH-`and` must be preferred to IC.

—So far, there is no surprise and the theoretical predictions are mostly confirmed; however, when the accessor simulation sequence is complicated by an extra `shift` (on processors where PH-`and+shift` differs from PH-`and`) or the `mod` latency, the overhead is no longer additive; an explanation may be that the sequence forms a bottleneck that blocks instruction-level parallelism, whereas the single coloring `load` can be done in parallel in most cases.

Overall, PH-`and` is better than expected for method invocation and subtype testing, so it should provide very high efficiency in JAVA virtual machines for implementing interfaces. When used for attribute access, the overhead becomes less reasonable. In contrast, the integer division overhead is higher than expected on many processors and it would seem that PH-`mod` should be reserved to processors that have efficient integer division—contrary to the processors tested here which uses the floating-point unit for integer division. Of course, the validity of such experiments that rely on a single program may be questioned. This is however a large program, which is fully object-oriented and intensively uses the basic mechanisms that are tested. Moreover, as the experiments compare two implementations with all other things being equal, the sign of the differences must hold for all programs and only the order of magnitude should vary. This limitation is also inherent to our experimentation. The PRM compiler is the only one that allows such versatility in the basic implementation of object-oriented programs. The counterpart is that the language has been developed with this single goal, so its compiler is the only large-scale program written in PRM.

4. CONCLUSIONS AND PERSPECTIVES

Our previous works on perfect hashing [Ducournau 2008] concluded that the technique was promising and deserved further consideration. However, a more in-depth assessment was also required. The tests presented in this paper allow us to draw some new and much more definitive conclusions about the time and space efficiency of perfect hashing.

From the space standpoint, we have proposed and tested, in this paper: (i) a family of hashing functions, (ii) an optimized approach, namely perfect numbering, and (iii) a more systematic testbed based on random class loading. It follows from these new tests that

—the tests presented in [Ducournau 2008] were optimistic because the class loading order was arbitrary and not representative of any possible class loading order;

—however, the conclusions concerning *modulus*-based perfect hashing still hold, as the variations according to the class loading order are not significant; on the contrary, PH-`and` appears to be erratic and over space-consuming in the worst-case class loading orders;

—anyway, *perfect class numbering* provides a marked improvement over plain perfect hashing, with both `mod` and `and` hashing functions; PN-`mod` is certainly the best tradeoff using `mod`;

—however, PN-`and` presents a dual face—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.

At this point, the conclusion would be to prefer PN-`mod` or PN-`and+shift`, with the time-space tradeoff that the former involves the latency of integer division whereas the latter remains slightly over space-consuming in the rare worst-case load orders.

However, these first conclusions rely on the assumption that all linear extensions represent possible class loading orders—this is certainly not the case and a better assumption would be that only concrete classes should be ordered. As our benchmarks do not include information about abstract classes, we considered a common assumption that only leaves are concrete. Under this assumption, all PH and PN functions are far better. Actually the erratic behavior of PH-`and` vanishes, especially with perfect numbering, so the conclusion would now be that PN-`and` is the solution.

Regarding time-efficiency, it is quite interesting to note that the conclusions are not halfhearted, namely PH-`mod` and PH-`and` are respectively worse and better than expected. Our experiments on the PRM compiler show that the overhead of *modulus* is marked on a processor which uses the floating-point unit for integer division, with a 20- or 25-cycle latency. Therefore, PH-`mod` should be reserved to processors with efficient integer division. On the contrary, the time-efficiency of PH-`and` is better than expected when used for method invocation and subtyping tests. Therefore, PN-`and` is certainly a very good solution for implementing JAVA interfaces—this is actually the best solution that we are aware of. Moreover, when it is also used for attribute access, the perfect hashing overhead becomes unreasonable—hence, it might be not an alternative to the C++ subobject-based implementation. However, the accessor simulation overhead has been overestimated in our tests—indeed, true accessors are also intensively used in the tested programs, in such a way that they add both overheads of accessor methods and simulation. So accessor methods should be implemented by direct access to the attribute, at least when the method is generated by the compiler.

So we can conclude this paper by recalling the five criteria that we stressed in [Ducournau 2008], namely (i) constant-time, (ii) linear-space, (iii) compatible with multiple inheritance, (iv) compatible with separate compilation and dynamic loading, (v) implemented with a code sequence that is short enough to be inlined. Coloring does not satisfy (iv) and C++ does not satisfy (ii). Moreover, there is no known constant-time technique for subtype testing that is directly compatible with the C++ implementation. Currently, all known JAVA interface implementations do not meet either (i) or (iv) requirements. For instance, the proposal of Palacz and Vitek [2003] is not inherently incremental, so it yields potentially high load-time overhead, together with extra run-time indirections. All other techniques, for instance [Alpern et al. 2001] or [Click and Rose 2002], are not time-constant. The only exception might be the proposal by Gagnon and Hendren [2001] of using direct access tables, that instead does not meet (ii). However, the empty slots of these huge tables are used for allocating other data, and we do not know the extent to which it counterbalances the nonlinear size—moreover, this trick cannot be used for subtype testing, for which empty entries represent failure, hence useful information. Overall, to our knowledge, perfect hashing is the only technique that fulfils all five criteria and the tests described in this paper show that its time constant and space factor are quite good.

The prospects of this work are manyfold:

—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 would be interesting to compare it with the C++ subobject-based technique which is the only one, with PH, that ensures constant-time method invocation in a multiple inheritance and dynamic loading framework, while being inherently incremental.

—As aforementioned, accessor simulation must be optimized in the case of true accessor methods—this might make PH competitive with subobject-based implementations.

—The first experiments in PRM led us expect that PH-and would be very efficient for implementing JAVA interfaces—so experiments in a production virtual machine would be very interesting in order to confirm it.

—Experiments with the PRM testbed must be generalized to other processor families, manufacturers or architectures—especially for testing more efficient integer divisions; other programs than the PRM compiler would also be of great interest.

—PN-and remains dependent on the class loading order; our assumption that only the leaf ordering matters should be confirmed, either empirically or theoretically. However, large-scale empirical experiments on real programs seem to be almost impossible. Palacz and Vitek [2003] present empirical experiments and they note that their benchmarks are running JAVA programs based on almost ten-thousand class libraries but that each run hardly loads one thousand classes and interfaces. Perfect hashing and perfect numbering with bit-wise and are always very good on one-thousand class benchmarks. So it would seem that the scalability of these techniques cannot be proven otherwise than by simulations. This is, however, somewhat unavoidable—scalability is the ability to gracefully scale up over future programs. 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"*.

APPENDIX

A. ABOUT PERFECT HASHING FUNCTIONS WITH BIT-WISE AND

The effects of bit-wise perfect hashing are not always intuitive, so we present here some very simple results that might help readers. Besides the intuition, perfect hashing presents a simple lower bound and a fine optimality condition. In the following, hashing means perfect hashing with bit-wise and. Let I be a set of n positive or null integers, m an integer that serves as a bit-wise mask and $H = m+1$ the hashtable size.

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

PROPOSITION A.1 (Bit-wise and lower bound). *Hashing $n > 0$ numbers requires a mask with at least $\log_2 n$ 1-bits, so $2^{\lceil \log_2 n \rceil} \in [n, 2n[$ 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 numbers. □

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. So, contrary to PH-mod, 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.

This certainly accounts for the different behaviors of both hashing functions and, in practice, it prevents us from using the same algorithms for both functions.

A mask m is *minimal* iff switching any 1-bit in m makes it loose its injectivity on the considered set.

PROPOSITION A.2 (Minimal mask 1-bit count upper bound). *A perfect hashing mask m that is minimal for a set I with cardinality n has at most $n-1$ 1-bits.*

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 numbers 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 agree with x on all 1-bits of m'. So 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 $I = \{2^i \mid i = 0..n-1\}$. $\square$

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

PROPOSITION A.3 (Minimal mask 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 bound $2\max(I) - 1$ can be reached.*

The minimal mask is bounded by the integer formed by all the 1-bits of all numbers in I. So an upper bound is formed by a chain of k consecutive 1-bits, where $k-1$ is the rank of the leftmost 1-bit in $\max(I)$. In the worst-case, all these 1-bits are required to form the mask—for instance, if $k = n$ and $I = \{2^i \mid i = 0..n-1\}$. $\square$

The next proposition is the basis for perfect numbering. It means that any set can be completed to reach a 2^k cardinality while keeping the same bit-wise mask.

PROPOSITION A.4. *Let I be a set of n numbers and m a mask that forms a perfect hashing function on I. Let k be the number of 1-bits of m. Then there is a set J of cardinality $2^k - n$, disjoint from I, such that m forms a perfect hashing function on $I \cup J$.*

This is a direct consequence of Proposition A.1. Among the 2^k combination of 1-bits of m, only n are used by I and the remaining is free and can be used for numbers in J. $\square$

So, in theory, it is always possible to allocate $2^k - n$ free numbers that fit the hashtable. However, in practice, class identifiers are bounded by the underlying integer implementation, e.g 16- or 32-bit integers, and it might happen that no free numbers fit the hashtable, though our experiments show that it would be very unlikely.

The last proposition is a limited converse of Proposition A.1, thus a condition for optimality of PN-`and`.

PROPOSITION A.5 (Optimality of PN-`and`). *Perfect numbering is optimal, i.e. $H_c = 2^{\lceil \log_2(n_c) \rceil}$, for every class c that is 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' its single direct superclass. 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, so an extra bit is required and $H_c = 2^{k+1}$. Otherwise, according to Proposition A.4, the mask has some place for an extra identifier and it can be inherited by c, hence $H_c = H_{c'} = 2^{\lceil log_2(n_c) \rceil}$. □

Intuitively, all 1-bits of the mask form a rightmost prefix of the mask and are "inherited". 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. So extra bits are required, that can be much more leftward, yielding an exponential increase in the mask. This explains why PN and PH can be erratic with bit-wise `and` in multiple inheritance. Proposition A.5 is important because all multiple inheritance class hierarchies are mostly in single inheritance—see, for instance, the statistics in [Ducournau 2006]. Hence, PN-`and` should be optimal on a large part of the hierarchies and this should counterbalance its worst-case behaviour on the multiple inheritance core. In contrast, PN-`and+shift` cannot improve on PN-`and` for all single inheritance classes—the shift is always 0—and the observed improvement only comes from the multiple inheritance core.

Finally, one observes that this single-inheritance optimum yields an overall ratio $\rho = \sum_c H_c / \sum_c n_c$ in interval $[1, 2[$. The upper bound is asymptotically reachable, that is, for all $\epsilon > 0$, there are class hierarchies such that $\rho > 2 - \epsilon$. Consider, for instance, a chain $A_{2^k} \prec A_{2^k-1} \prec ... \prec A_1$ 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 $H_c = 2^{k+1}$. So, in this framework, x can be large enough to make $\rho \approx 2^{k+1}/(2^k+1)$. In practice (Table II), ρ is always lower than 1.5 in our multiple inheritance benchmarks.

B. ALGORITHMS

The computation of perfect hashing parameters is presented in a simple COMMON LISP code [Steele 1990], like in PHAPST. We first present the code for PH-`mod` then PH-`and` and its variants. In the following algorithms, we give first priority to simplicity over efficiency.

B.1 With Modulus

The general idea for computing the PH parameter is to test each number, beginning from the cardinality of the set that must be hashed. This works for both `and` and `mod` hashing functions, but it generalizes to PN only for `mod`. So, with `mod`, the functions for both perfect hashing and perfect numbering are built according to this common schema and the latter is just completed by the allocation of free IDs.

The input of the basic function is a list `ln` of all-different integers—the identifier of superclasses that are already loaded—plus a number `n` of new IDs that must be allocated. The output is the H parameter, i.e. the least integer greater or equal to the total number of identifers, that makes `mod` injective on the list of already known identifiers.

```
(defun basic-ph-mod (ln n)
  (if (null ln)
      (max 1 n)
    (loop for h from (+ (length ln) n) by 1
          when (ph-p-mod ln h) return h)))
```

The `ph-p-mod` function checks that its parameter `h` forms a perfect hashing function for the list `ln` of identifiers. `*ht*` is a global variable that is bound to an array that is presumed to be large enough. When `ph-p-mod` succeeds, its call leaves the hashtable filled by numbers in `ln` on length `hc`.

```
(defun ph-p-mod (ln h)
  (fill *ht* nil :start 0 :end h)                  ;; resets *HT*
  (loop for i in ln
        for hv = (mod i h) do
        (if (aref *ht* hv)
            (return nil)                           ;; fails
          (setf (aref *ht* hv) t))
        finally return t))                         ;; succeeds
```

The `ph-mod` function is a simple call to the base function:

```
(defun ph-mod (ln) (basic-ph-mod ln 0))
```

Perfect numbering has two parameters, the set `ln` of superclass IDs and the number `n` of new IDs that must be allocated. There is actually no need to make a difference between the case where all superclasses are already loaded (so, the number is 1) and the case where a set of classes or interfaces is loaded as a whole. The function begins by computing the H_c parameter, by calling `basic-ph-mod`, then selects the first free numbers that are hashed in an empty entry. It returns two values[5], namely the set of new identifiers `idc` and the hashtable size `hc`.

```
(defun pn-mod* (ln n)
  (let ((hc (basic-ph-mod ln n)))
    (values (compute-least-free-ids-mod n hc) hc)))

(defun compute-least-free-ids-mod (n hc)   ;; computes N free identifiers
  (let ((idc))
    (loop for i in *free* until (= n 0)
          for hv = (mod i hc) do
          unless (aref *ht* hv) do
          (setf (aref *ht* hv) t)
          (push i idc)
          (decf n))
    idc))
```

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

`*free*` is a data structure that represents the set of free identifiers. Note that 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. More efficient algorithms are possible, for instance one could start from the empty entries in `*ht*` and search for their intersections modulo `hc` with intervals in `*free*`. It would certainly improve the worst-case complexity but our tests show a marked degradation on average. However, the naive loop is efficient in practice and can also be applied to PN-`and`.

B.2 Basic Perfect Hashing With Bit-wise `and`

PH-`and` can be based on exactly the same function, analogous to `basic-ph-mode` but, in the case of bit-wise `and`, it only works for $n = 0$ and does not generalize to perfect numbering. This is the consequence of Proposition A.1—with bit-wise `and`, a hashtable can have empty entries that cannot be filled by any number. So an algorithm is required that works at the bit level. By the way, it is also more efficient as it yields a logarithmic complexity. For the basic version of PH-`and`, one first computes a mask with all discriminant bits, i.e. bits which 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)
  (if (null (cdr ln))
      1
    (let ((mask (logxor (apply #'logior ln) (apply #'logand ln))))
                                        ;; MASK consists of all discriminant bits
      (loop for b from (1- (integer-length mask)) by 1 downto 0
            when (logbitp b mask) do              ;; B-bit is 1 in MASK
            (let ((new (logxor mask (ash 1 b))))  ;; NEW is MASK with B-bit at 0
              (when (ph-p-and ln new) (setf mask new)))
            finally return (1+ mask)))))
```

In the previous code, `logxor`, `logior` and `logand` are Common Lisp integer functions for bit-wise operations: exclusive and inclusive `or`, and `and`. `(Integer-length n)` gives the position of the leftmost 1-bit of a positive integer `n`. `(Logbitp n b)` tests if the `b`-th bit of `n` is 1. `(Ash n b)` shifts `n` left by `b` positions (when `b` is positive). Function `ph-p-and` is the analogue of `ph-p-mod` for bit-wise `and`.

```
(defun ph-p-and (ln mask)
  (fill *ht* nil :start 0 :end (1+ mask))              ;; resets *HT*
  (loop for i in ln
        for hv = (logand i mask) do
        (if (aref *ht* hv)
            (return nil)
          (setf (aref *ht* hv) t))
        finally return t))
```

Like `ph-p-mod`, when `ph-p-and` succeeds, its call leaves the hashtable filled by numbers in `ln` on length `hc`. However, contrary to `ph-mod`, `ph-and` does not finish by a successful call to `ph-p-and`, therefore a subsequent call to `ph-p-and` may be required in the following functions.

B.3 Perfect Class Numbering

PN-and is slightly more complicated than both PH-and and PN-mod. The algorithm is based on Proposition A.4 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-p-and and the function ends like pn-mod*, by selecting free identifiers and returning 2 values.

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

(pn-and* ln n) is certainly not exactly equivalent to iterating (pn-and ln)= (pn-and* ln 1) n times. Their respective results must depend, however, on the structure of the class hierarchy. Consider, for instance, a diamond formed by classes A, B, C and D such that $D \prec B, C \prec A$. Suppose that only A is already numbered, so all other classes must be numbered as a whole (i.e. n=3). Proposition A.5 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. So, 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. Finally, a mixed algorithm has been implemented. The idea is to use the iterative form for all single inheritance trailing chains. Suppose that the diamond is extended by two chains $A = A_m \prec A_{m-1} \prec .. \prec A_1$, and $D_p \prec D_{p-1} \prec .. \prec D_1 = D$, and all of these classes must be numbered as a whole when D_p is loaded. The chain $(A_1, A_2, .., A_m)$ is first numbered according to the iterative scheme, then the rest of the diamond is globally numbered and, finally, the chain $(D_2, .., D_p)$ is iteratively numbered. This general algorithm also applies to PN-mod, though we do not have any optimality result. However, in practice, it does not yield markedly better results and the data in Tables IV and V are produced by pn-and*.

```
(defun compute-least-free-ids-and (n mask)
  (let ((idc))
    (loop for i in *free* until (= n 0)
          unless (aref *ht* (logand i mask)) do
          (setf (aref *ht* (logand i mask)) t)
          (push i idc)
          (decf n))
    idc))
```

Optimizing the loop in compute-least-free-ids-and is likely not as straightforward as for compute-least-free-ids-mod. Determining empty entries in *ht* first requires the combinatorics of all 1-bits in mask. However, it seems to be difficult to compute the least number in an interval that matches one of these free entries without enumerating all numbers in the interval. So this naive loop is likely a good tradeoff between simplicity and efficiency. On the other hand, the algorithm could be slightly changed in order to compute the least n integers that fit the least n empty entries in the table.

B.4 Bit-wise shift

The combination of and and shift relies on a similar algorithm but the optimal value is less straightforward, so we use nonoptimal heuristics and only present one of them. Anyway, the function must return four values—namely the hashtable size, the bit-wise mask, the shift (i.e. the rightmost 1-bit) and the leftmost 1-bit. The function is like ph-and except that the 1-bits of the mask are not scanned in the same order. It first attempts to switch leftmost 1-bits, then rightmost 1-bits, and finally all remaining bits from left to right.

```
(defun ph-and+s (ln)
  (if (null (cdr ln))
      (values 1 0 0 0)
    (let ((mask (logxor (apply #'logior ln) (apply #'logand ln)))
          (lbit (1- (integer-length mask)))              ;; leftmost 1-bit
          (rbit))                                         ;; rightmost 1-bit
      (loop for lb from lbit by 1 downto 0                    ;; leftmost
            when (logbitp lb mask) do                         ;; 1-bit
            (let ((new (logxor mask (ash 1 lb))))             ;; switch
              (if (ph-p-and ln new)
                  (setf mask new)
                (return (setf lbit lb)))))
      (loop for rb from 0 by 1                                ;; rightmost
            when (logbitp rb mask) do                         ;; 1-bit
            (let ((new (logxor mask (ash 1 rb))))             ;; switch
              (if (ph-p-and ln new)
                  (setf mask new)
                (return (setf rbit rb)))))
      (loop for b from (1- lbit) by 1 downto (1+ rbit)      ;; left to right
            when (logbitp b mask) do                        ;; inner 1-bit
            (let ((new (logxor mask (ash 1 b))))            ;; switch
              (when (ph-p-and ln new) (setf mask new))))
      (values (1+ (ash mask (- rbit))) mask rbit lbit))))
```

The combination with perfect numbering gives the following algorithm. The main difference with pn-and* is that the bit that is switched is first selected in the interval 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 rbit lbit) (ph-and+s ln)
    (loop while (> (+ (length ln) n) (ash 1 (logcount mask))) do
          (loop for b from rbit by 1                      ;; rightmost
                unless (logbitp b mask) do                ;; inner 0-bit
                (when (and (> b lbit) (> rbit 0))
                   (setf b (1- rbit)
                         rbit b))
                (setf mask (logxor mask (ash 1 b))          ;; switch
                      hc (1+ (ash mask (- rbit))))
                (return)))
    (ph-p-and ln mask)                                      ;; ends like PN-AND*
    (values (compute-least-free-ids-and n mask) hc)))
```

REFERENCES

Alpern, B., Cocchi, A., Fink, S., and Grove, D. 2001. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proc. OOPSLA'01*. SIGPLAN Notices, 36(10). ACM Press, 108–124.

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

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

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

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

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

Ducournau, R. 2006. Coloring, a versatile technique for implementing object-oriented languages. Rapport de Recherche 06-001, LIRMM, Université Montpellier 2.

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

Ducournau, R. 2009. Implementing statically typed object-oriented programming languages. *ACM Computing Surveys*. (to appear).

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

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

Lorenz, M. and Kidd, J. 1994. *Object-Oriented Software Metrics.* Prentice-Hall, Englewood Cliffs (NJ), USA.

Meyers, S. 1996. *More Effective C++*. Addison-Wesley.

Morandat, F., Ducournau, R., and Privat, J. 2009. Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique. In *Actes LMO'2009*, B. Carré and O. Zendra, Eds. Cépaduès, 17–32.

Morris, R. 1968. Scatter storage techniques. *Commun. ACM 11,* 1, 38–44.

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

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

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

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

Steimann, F. 2000. Abstract class hierarchies, factories, and stable designs. *Commun. ACM 43,* 4, 109–111.

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