Perfect Hashing for Method Dispatch with Dynamic Typing and Dynamic Compilation

Roland Ducournau
LIRMM – CNRS and Université Montpellier II, France
RR LIRMM RR-12010

April 2, 2012

Abstract

In static typing, the receiver's static type is the key to efficient implementation of method invocation, and a recently proposed technique, based on perfect hashing of classes, cannot apply to dynamic typing because of the lack of static types. In this article, we propose a new application of perfect hashing to method dispatch in a dynamic typing, dynamic loading and single inheritance setting. The approach involves hashing method selectors instead of classes. However, as hashing all methods revealed itself to be space-inefficient, only *overloaded* methods, ie methods introduced by several classes, are hashed. The dispatch of non-overloaded methods is done as in *single-subtyping*, ie static typing and single inheritance.

An adaptive-compilation protocol and an algorithm for hashing overloaded methods are proposed, and the approach is tested on SMALLTALK benchmarks by simulating class loading at random.

Keywords: method dispatch, single inheritance, dynamic typing, object representation, compilation protocol, perfect hashing, perfect numbering, adaptive compiler.

1 Introduction

In spite of its 30-year maturity, object-oriented programming still has a substantial efficiency drawback in the *dynamic loading* context and it is worsened by either *multiple inheritance*, or *dynamic typing*. In the dynamic loading context, compilation must be *separate* (as opposed to *global*), ie code units are compiled separately from each other. Compilation can also be *static* or *dynamic*. With static compilation, each code unit is compiled once for all, and this implies the *open-world assumption* (OWA) which makes the generated code rather inefficient. In contrast, with dynamic compilation, a code unit can be further recompiled; it thus allows the compiler to perform aggressive optimizations, based on provisory *closed-world assumptions* (CWA), which makes the code more efficient, at the price of extra recompilations. The overall efficiency is thus a tradeoff between the runtime efficiency of the generated code and the recompilation cost. This article focusses on both dynamic compilation and dynamic typing.

In recent articles, we proposed a new implementation approach, called *perfect class hashing* [1] and based on *perfect hashing*, ie truly constant-time, collision-free hashing [2, 3]. From an algorithmic point of view, a variant called *perfect numbering* involves optimizing the class IDs in order to minimize the hashtable sizes [4]. Real-size experiments in the PRM compiler [5] showed that perfect hashing would be quite efficient for implementing *multiple subtyping*, ie JAVA interfaces, in a static, separatate compilation setting. However, it would be rather inefficient, ie as inefficient as C++-like subobjects, when used in a full multiple-inheritance context [6]. Therefore, we also proposed a compilation/recompilation protocol that would allow for an efficient implementation in a *just-in-time*, dynamic compiler [7]. Indeed, in this dynamic setting, most invocation sites can use shortcuts that are more efficient than perfect hashing.

However, perfect class hashing can be used only in a *static typing* setting, because it involves hashing classes, and grouping methods by introduction class. On a given invocation site, the introduction class is thus deduced from the receiver's static type. In contrast, even with single inheritance, in a *dynamically typed* language like SMALLTALK [8], a method (ie a method *selector* in SMALLTALK jargon) may be introduced by several classes. Hereafter, we will say that such methods are *overloaded*[1]. Therefore, the perfect hashing approach requires, in this new context, an efficient way of hashing methods instead of classes, and first experiments showed that perfect method hashing was not that space-efficient [1].

In this paper we propose an object representation and a recompilation protocol which provide an efficient use of perfect method hashing for implementing SMALLTALK-like languages. The main idea is that perfect hashing is used only for overloaded methods, which are presumed to be few enough for keeping the overhead low. However, as the overload feature depends on the classes that are actually loaded, these hashtables must be recomputable, and an extra indirection is required. In contrast, methods that are introduced by a single class are invoked in the same way as in *single-subtyping* (SST), by deducing the single introduction class from the method selector instead of the receiver's static type. Besides, common optimizations like monomorphic invocations can be considered.

Plan. The rest of the article is structured as follows. The next section presents the object-representation issue, and states our proposal of applying perfect hashing (more precisely, a variant called *perfect numbering*) to overloaded methods. Section 3 presents the compilation/recompilation protocol the technique requires in an adaptive compiler. Section 4 recalls the perfect hashing problem, formalizes the new algorithmic problem and proposes a heuristic which is, in some sense, optimal, though non-optimal on many aspects. Section 5 presents our experiments, based on a simulation of large benchmarks with random class-loading. These tests show that overloaded methods are few enough, thus making the overhead of perfect hashing very low. Furthermore the algorithm appears to

[1]This use of the *overload* term must not be confused with *static overloading*, which represents, in languages like C++, C# or JAVA, the fact that a method name can be used in the same context with different parameter types or numbers. It must not either be confused with *overloaded functions* which represent methods dispatched on all their parameters [9]. Finally, it must not be confused with *overriding*, which represents the fact that a method defined in a class can be redefined in a subclass.

be space-efficient enough to be further considered. Conclusions and prospects end the article.

2 Object and value representation

2.1 Object representation

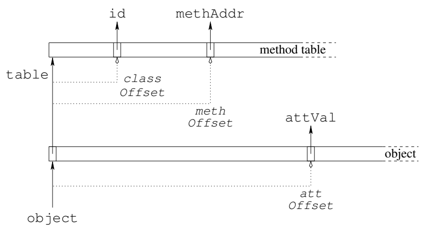
Single subtyping. Figure 1 describes the implementation used in single-subtyping (SST), ie in static typing, when all types are classes and have a single direct supertype. This implementation is simple and efficient, as it verifies the *position invariant*: the considered method or attribute is always implemented at the same position, whatever the receiver's dynamic type. In this implementation, methods and attributes are grouped by introduction class (type) and the group of a class is appended to the structure of its direct superclass. Subtype testing is implemented with the technique known as *Cohen's display* [10], with the class ID in each method group. However, the SST approach cannot be used either with multiple inheritance or dynamic typing.

Perfect class hashing in static typing. Figure 2 describes the implementation of method invocation and subtype testing with perfect class hashing, in a static typing setting. It can be used directly for multiple subtyping (ie JAVA interfaces). In the method tables, the positive offsets contain the SST implementation which is used for class-typed invocations. It can also be extended to attribute access for full multiple inheritance, but this extension (called accessor simulation) is not that efficient. In [7], we proposed to used this implementation in a dynamic loading setting, in conjunction with a recompilation protocol which allows the compiler to shortcut most of the polymorphic PH invocations with SST invocations. This optimization is possible when the position invariant holds, and it is made efficient by the fact that it holds most of the time. Of course, all monomorphic method calls[2] are shortcut with static calls. It would thus avoid most of the uses of PH in actual invocations. Anyway, this approach cannot work with dynamic typing.

All-method perfect hashing in dynamic typing. Figure 3 describes the implementation of method invocation and subtype testing in dynamic typing, with perfect method hashing. In single inheritance, it would be more efficient to remove class IDs from the hashtable and put them as in SST but in negative offsets. In multiple inheritance, the hashtable must be extended for attribute access, with accessor simulation, but it works only with SMALLTALK-like encapsulation. This new approach would work with dynamic typing but it would not be that efficient. Indeed, with this implementation, there is no way to shortcut polymorphic PH invocations with a more direct invocation sequence, because the position invariant does not hold.

[2]An expression is said to be monomorphic when its value at runtime will always be of the same dynamic type. When the receiver is monomorphic, a virtual call always invoke the same method. Here, we use monomorphic in a wider meaning, that is when a virtual call always invoke the same method (even when the receiver is not monomorphic).

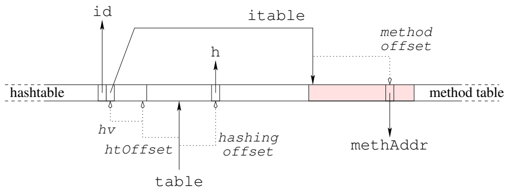
```
// attribute access
load [object + #attOffset], attVal

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

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



Code sequences for the 3 basic mechanisms and the corresponding diagram of object layout and method table. The pseudo-code is borrowed from [11]. Pointers and pointed values are in Roman type with solid lines, and offsets are italicized with dotted lines.

Figure 1: Single-subtyping implementation

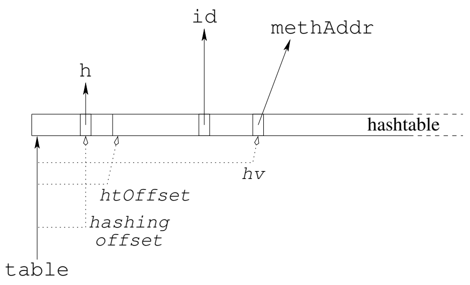
```
//preamble for both mechanisms
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #interfaceId, h, hv
sub table, hv, htable

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

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



The method table is bidirectional. Positive offsets contain method addresses and class IDs, as in SST, and negative offsets consist of the hastable, with a twofold entry for each implemented interface. The grey rectangle denotes the group of methods introduced by the considered interface. `fieldLen` represents the entry size, e.g. 8 if 32-bit integers are used. In practice, all numbers (i.e. H and class ID's) must be multiplied by `fieldLen` (of course, it works only if it is a power of 2).

Figure 2: PH for JAVA interfaces

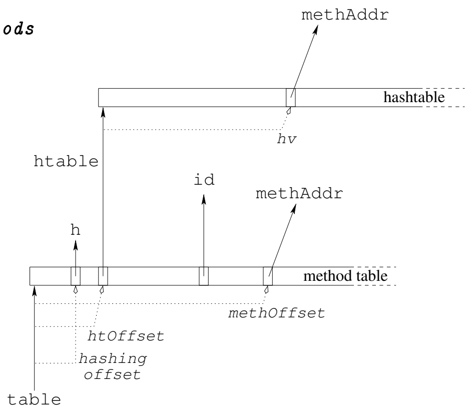
```
//method invocation
load [object + #tableOffset], table
load [table + #hashingOffset], h
move #methId, methId
and methId, h, hv
add table, hv, htable
load [htable +#htOffset], methAddr
call methAddr
```



The method table consists of the hashtable, which contains a single entry per method or class. With dynamic typing, the method identifier (`methID`) must be checked in the method prologue, unless the method has been introduced by the hierarchy root.

Figure 3: PH of all methods for dynamic typing and multiple inheritance

```
// method invocation for overloaded methods
load [object + #tableOffset], table
load [table + #hashingOffset], h
load [table +#htOffset], htable
move #methId, methId
and methId, h, hv
add htable, hv, htable
load [htable], methAddr
call methAddr

// other invocations are like
// with SST
```



The method table is the same as with SST, apart from two extra fields for referencing the hashtable (`#htOffset`) and the hashing parameter (`#hashingOffset`), which can be loaded in parallel on processors that provide instruction-level parallelism. The hastable contains a single entry per method, only for methods that have been introduced by more than one class.

Figure 4: PH of overloaded methods for dynamic typing and single inheritance

PH restricted to overloaded methods. Finally, Figure 4 presents the implementation we propose in a SMALLTALK-like setting. Like the multiple-subtyping implementation (Figure 2), it combines the SST implementation with a hashtable. However, instead of being inlined in the negative offset of the method table, the hashtable is now a separate table, and is thus accessed via an indirection. This is of course less efficient, mostly because of ensuing cache-misses, but the hashtable is now restricted to methods that are introduced by several classes, and it should be used only marginally.

2.2 Special cases

Besides this general object representation, dynamic typing yields a few specific cases which complicate the generated code and add overhead. These points are irrelevant for attributes when they are encapsulated, ie accessible only on `self`, as in SMALLTALK.

2.2.1 Unknown-method exception

In dynamic typing, the compiler cannot ensure that the receiver knows the invoked method. A dynamic check is thus necessary, in most cases. When the method address is gotten from the hashtable, it may result from an empty entry, and instead of testing that this entry is empty, it is better to fill reachable, empty entries with the address of a function which signals the *unknown method* exception[3]. When an actual method address is gotten from the hashtable or the SST method table, it may correspond to another method selector, instead of the invoked one. Therefore, a test must be made at the method entrypoint,

[3]In SMALLTALK, a `doesNotUnderstand` message is sent to the receiver.

```
// method invocation with per-method tables
cmp object, #0
bgt #vft
and  object, #tagMask, tag
rshift tag, #tagShift, tag
add tag, #methAddr, ttable
bra #common
vft:
load [object + #tableOffset], table
load [table + #hashingOffset], h
load [table +#htOffset], htable
move #methId, methId
and methId, h, hv
add htable, hv, ttable
common:
load [ttable], methAddr
call methAddr
```

```
// method invocation with PIC
cmp object, #0
bgt #vft
and object, #tagMask, tag
rshift tag, #tagShift, tag
cmp tag, #expectedTag
bne #othertag
call #expectedMeth
bra #end
otherTag:
// another test
vft:
// usual sequence for objects
end:
```

Figure 5: Method invocation for primitive values

and the best way to do it is to pass the method selector in a register and test it against the expected selector as the first instruction of the method body[4].

2.2.2 Primitive values

In dynamic typing, the type of values must be encoded in the value itself, and it would be quite inefficient to use, for integers or characters, the same representation as for true objects. Therefore the type is encoded in the bit-representation of the value, instead of being encoded in the method table. A practical solution, for instance, involves reserving the leftmost byte as a type-tag, with a leftmost bit at 1. Hence, a primitive value is a negative number, whereas a reference is positive.

Then a method invocation sequence begins with testing whether the receiver is tagged, ie a negative number, then extracts the tag with a bit-wise mask and a shift. Several alternative techniques can then be used, and they can be mixed. A method table can be associated with each primitive type, and accessed via a table of tables, with the tag as an offset. Conversely, a table can be used for each method, with an entry for each primitive type (Figure 5, left). When the number of expected tags is low, especially on processors that are equipped with conditional branching prediction (ie all processors except those that are dedicated to small embedded systems), the technique known as *polymorphic inline cache* (PIC) [12] is often preferred. The extracted tag is now tested for equality against each expected tag, and the right method is then statically called (Figure 5, right). The PIC sequences for a given method selector could be factorized in a common stub function. It would improve the code size, to the detriment of runtime efficiency, because branch prediction is markedly more accurate when PICs are inlined [5].

The tag test can be saved on only in two specific situations: (i) when the receiver is `self`, and the enclosing class is either a specific primitive type, or a class without primitive subtypes; (ii) when the receiver is a literal or the direct

[4]In some cases where the test is useless, eg for monomorphic `self`-invocations (see hereafter), multiple entry points might save on this test, but this is a minor optimization since branch prediction would almost always succeed.

result of an instantiation (`new`). Therefore, apart from these situations, when the invoked method is not known by primitive types the code sequence from Figures 1 and 4 must begin with the following test

```
cmp object, #0
bgt #fail       //unknown-method exception
```

In contrast, in common superclasses of primitives types and usual classes, where `self` can be either a tagged value or a true object, method specialization is a way to avoid a long code sequence.

2.2.3 `null` value

Unitialized variables and attributes represent a constant issue in object-oriented programming. Initializing them with a distinguished `null` value is a first step, but ubiquitous `null`-checks would be inefficient. In the dynamic typing context, using 0 as the `null` value allows for integrating the `null`-check in the tag tests at minor expense. `null` then implies the 0 tag. When the invocation site cannot imply any primitive type, the prologue test has just to be replaced with the following

```
cmp object, #0
bge #fail       //unknown-method or null exception
```

3 Compilation protocol

As explained in the introduction, we are concerned, here, with dynamic compilation. The compilation/recompilation protocol is in charge of a few tasks: (i) computing the data structures associated with a class; (ii) generating machine-code from method source code (or bytecode); (iii) recompiling some data structures and pieces of code when the assumptions supporting the previous compilation are no longer valid.

The protocol relies on the principle that some part of the generated code or structure is compatible with the *open world assumption* (OWA), so that it is computed in an incremental way, once for all. In contrast, another part makes provisory *close world assumptions* (CWA), which allows the compiler to perform aggressive optimizations at the price of potential recompilations when underlying assumptions are refuted.

In the recompilation protocol, we consider dynamic class loading, but exclude class reloading.

3.1 Data structures

The data structures associated with a class are twofold. Each time a class is loaded, the runtime system builds a model of this class, ie an instance of some metamodel (eg the metamodel proposed in [13]). This model links the newly loaded class with its superclasses and the methods the class *defines*, *inherits* or *introduces*—these terms have intuitive meanings that are formalized in [13]. Methods that are introduced by several classes are of course distinguished, and we call them *overloaded.* This model is incremental and must be updated with further class loadings, when a previously non-overloaded method becomes overloaded. Non-overloaded is thus a provisory, mutable feature, while overloaded is definitive and immutable. Moreover, the model must maintain information about the liveness of classes and methods. A class is alive if it has been instantiated, while a method is alive if it has been already invoked. The definition may be slightly enlarged, by considering that compiling an instantiation or invocation site is enough for making a class/method alive. Therefore, the overall protocol can be considered as a kind of static analysis like *Rapide Type Analysis* (RTA) [14], which is run very dynamically as the classes are loaded and the code is executed. In this situation, RTA is equivalent to *Class Hierarchy Analysis* (CHA) [15]. Note, however, that this runtime analysis is different from profiling and remains static.

Class loading is triggered in two situations: (i) directly, when the considered class is instantiated (`new`); (ii) recursively, when a subclass is loaded. Methods tables are computed as soon as the considered class is instantiated, hence only in case (i), or when an already loaded class is instantiated for the first time. The method table itself is computed as in SST, but the hashtable is allocated in a lazy way, when it contains an overloaded method that is already alive, ie such that an invocation site has been already compiled. In the following, we will assume that this computation is done at class loading, but it could be postponed as well.

When the hashtable is computed, a perfect numbering algorithm is applied as follows. Let C be the considered class. Then M_C is the set of methods known by C that are introduced by several already-loaded classes. Some of these methods have already an ID, because they have been hashed in other classes. Let M'_C be this subset, and $I'_C = \{id_x | x \in M'_C\}$ be the corresponding set of method IDs. The other methods in M_C have no IDs, because they have not been hashed yet, and some subset M''_C must be hashed now. Note that the latter subset may include all the overloaded methods known by C, or only those that are invoked in some methods that must be compiled or recompiled; this is a matter of tuning of the protocol. The respective cardinalities of these sets are denoted n'_C and n''_C.

Perfect numbering is then applied with I'_C and n''_C as inputs; its outputs are a hash parameter H_c and a set of method IDs I''_C, of cardinality n''_C, which are assigned to the methods in M''_C. A hashtable of size H_C is computed, filled and linked to the method table.

The algorithmic aspect is developed in Section 4.

3.2 Code generation

In the following, we consider only method invocation. Indeed, the case of subtype tests is similar to, though simpler than, method invocation and we don't develop it. Besides, with SMALLTALK-encapsulation and single inheritance, attribute access is like with SST.

Method compilation is lazy, and it may be triggered by a trampoline in method- or hash-tables. When a method is compiled, each invocation site in the method body is compiled, and this is the main focus of the compilation protocol.

Specific cases. Firstly, different kinds of invocation sites can be distinguished from each other, and the distinctions can be static and hold under the OWA, or dynamic and hold under a provisory CWA. The distinction concerns both the receiver and the invoked method.

- Statically, the receiver may be a literal; `self`, ie the current receiver which is statically typed by the enclosing class; or anything else.
- Dynamically, the receiver may be `null`, a tagged value or a standard object.
- Statically, the invoked method may be introduced by the hierarchy root.
- Provisorily, the invoked method may be unknown, because the classes that introduce it have not been loaded yet, or it may be unknown in the static type of the receiver, in the specific case of `self`, or even in the receiver's dynamic type when it can be statically infered, eg for a literal or a direct instanciation (`new`).
- Provisorily, the invoked method may be monomorphic, it can be introduced by a single class, or several ones.
- Provisorily, the invoked method may be introduced by "standard" classes, ie classes without primitive subtypes, or not.

Decision tree. The complete combination gives the following decision tree whose conditions are tested sequentially (ie each condition implies the negation of the previous ones).

1. The receiver is a literal
   (a) the method is not known by the literal type → static type error;
   (b) otherwise → static call to the specific method (no test needed);
2. The receiver is `self`, and let C be the enclosing class, then
   (a) *the method is unknown* by all loaded subclasses of C → static call to the *unknown-method* function;
   (b) the method is known by C
      i. and *not redefined in the subclasses* of C → static call to the method inherited by C (no test)
      ii. otherwise
         A. C is a standard class → SST implementation
         B. C is a primitive type → PIC implementation
         C. otherwise → combined PIC-SST implementation
   (c) *the method is introduced by a single subclass D* of C,
      i. and *not redefined in the subclasses of D* → static call with a subtype test (depends on whether D is a standard class or a primitive type);
      ii. D is a standard class → SST implementation
      iii. D is a primitive type → PIC implementation
      iv. otherwise → combined PIC-SST implementation

   (d) the method is introduced by multiple subclasses of C
      i. the introduction subclasses are only standard classes $\rightarrow$ PH implementation
      ii. *the introduction subclasses are only primitive type* $\rightarrow$ PIC implementation
      iii. otherwise $\rightarrow$ combined PIC-PH implementation

3. otherwise
   (a) *the method is unknown* $\rightarrow$ static call to the *unknown-method* function;
   (b) the method is introduced by the root:
      i. *it is currently monomorphic* (not redefined in any class) $\rightarrow$ static call;
      ii. it is not redefined in primitive types $\rightarrow$ SST implementation;
      iii. otherwise $\rightarrow$ PIC+SST implementation;
   (c) *the method is introduced by a single class* $D \Rightarrow$ (2-c);
   (d) otherwise $\Rightarrow$ (2-d);

3.3 Recompilation protocol

As mentioned above, the protocol is based on a model of the programs which involves an explicit representation of classes, method selectors and definitions, and method invocation sites, along with their relations to each other. Each method selector memorizes the method invocation sites compiled in a provisory way, so that the loading and compilation of a new definition for this method selector can trigger the recompilation of the concerned sites or enclosing methods.

Technically, recompilation can work at the method level, or at the invocation-site level. In the former case, the whole method is recompiled and the content of some method tables is updated. In the latter case, the invocation site is compiled into a stub function, called a *thunk*, and the original method code is modified in order to call it. A mixed approach should likely be preferred. All this is discussed in [7], but it remains out of the scope of this paper.

3.4 Optimization, laziness and efficiency assessment

The optimization problem is actually markedly more complicated than stated in Section 3.1, because several hashtables must be optimized at the same time (one for each class introducing a method in M''_C), whereas what we called perfect numbering is intended to optimize a single hashtable. A more accurate formulation is presented in Section 4.

Efficiency assessment is characterized by three non-independent parameters, namely (i) the memory occupation, especially the hashtable size; (ii) the recompilation cost, for instance the number of hashtable allocations; (iii) the time-efficiency of the generated code. In the following, we will consider only the first two criteria, hashtable size and allocation number.

The overall protocol is essentially lazy, and laziness concerns not only allocation and computation time, but also computation content. Laziness should have marked impact on the two efficiency criteria, but it might be in opposite directions.

- Hashtables should be allocated just-in-time, hence only when an overloaded method is invoked on an instance of the considered class. A simple way to do it is to initialize each method table with a common single-entry hashtable filled with a trampoline which will allocate the actual hashtable. This is the only point for which there is no doubt, and just-in-time allocation will be optimal on all criteria.

- The actual hashtable computation, which involves assigning IDs to method selectors, could occur at any time between the class loading and the hashtable allocation. From both the hashtable-size and compilation-cost standpoints, the effect of the computation time is unclear, because our actual optimizing algorithm iterates on method selectors in a non-optimal way.

- When a hashtable is computed/allocated, it may consider the whole set of known overloaded methods, or only those that have been considered alive, eg because an invocation site has already been compiled. If computation and allocation are restricted to live methods, the hashtable size will be optimized. However, it is likely that many hashtables will be recomputed and reallocated a number of times, each time a new overloaded method becomes alive, thus increasing the recompilation costs.

4 Problem definition and algorithms

4.1 Notations

Set and function notations. We mostly consider integers and integer sets. Notations are as usual. Function notations are extended to sets in the usual way, ie $f(E) = \{f(x)|x \in E\}$. Usual operations are extended to sets in a similar way, namely $x + A = \{x + y|y \in A\}$ and $A + B = \{x + y|x \in A, y \in B\}$.

Notations for bitwise representation. The hashing function we use is the bitwise `and` operation. Hence bitwise representation is essential, but it is often more intuitive to use instead a set-wise representation.

Let k be an integer number. Then, $b(k)$ denotes the set of 1-bits of k, ie the set of their positions in k. Given two numbers j and k, $j \subset k$ is a shortcut for $b(j) \subset b(k)$. Therefore a number can be represented as a set of number (its 1-bit positions) and can represent another set of numbers that have a subset of its 1-bits (however the notation will be $j \subset b(k)$ instead of $j \in b(k)$). All these sets are of course finite, and $b^{-1}(E)$ is defined for any finite set $E \subset \mathbf{N}$.

In order to remove any ambiguity, given a set of numbers E, $[E]$ will denote the set of numbers whose 1-bit positions are included in E, ie $[E] = \{n \in \mathbf{N}|b(n) \subset E\}$.

Moreover, $\overline{k}$ represents the complement of k, ie $\overline{k} = \mathbf{N} \backslash b(k)$. Note that $\overline{k}$ does not represent a number, because it is an infinite set. Given two numbers j and k, $j \subset \overline{k}$ is a shortcut for $b(j) \subset \overline{k}$ (or, equivalently for $j \in [\overline{k}]$). Other set notations will be used in a similar way.

Finally, we will use bitwise boolean operations `or` and `and`, with the usual logical notations $\vee$ and $\wedge$. For instance, $j \vee k = b^{-1}(b(j) \cup b(k)) = \texttt{or}(j,k)$.

4.2 Problem definition

4.2.1 Problem input

1. X is a class set,
2. each class $x \in X$ is associated with a set I_x of integers, and a set S_x of symbols;
3. p is a distinguished element in X, and $S_p = \bigcup_x S_x$;
4. for each $s \in S_p$, $\exists x \in X$ such that $x \neq p$ and $s \in S_x$.
5. finally, F is a set of free identifiers, ie $F \subset \mathbf{N}$ and $F \cap I_x = \emptyset$ for each $x \in X$.

Perfect hashing is applied to the I_x sets with the bitwise `and` hashing function. Therefore, for each $x \in X$, $m_x = pha(I_x)$ is the least integer such that the function $y \mapsto \texttt{and}(m_x, y)$ is injective on I_x. See [1, 4] for more details on this use of perfect hashing.

Context. X is a set of classes in a dynamic typing and dynamic loading setting. p is the class that is currently loaded and compiled in a lazy mode, and S_p is the set of methods introduced by p that are also introduced by other classes. Therefore, X depends on p, and can be defined as the set of already loaded classes that introduce a method that is also introduced by p. Classes in X are loaded but may not be compiled yet.

Note that there are definitely no constraints between I_x sets, since they may as well be disjointed, overlap or contain each other. The same is true with S_x sets, apart from points (3) and (4) above.

4.2.2 Problem output

The problem is to assign a unique identifier to each $x \in S_p$, more formally to define an injective function $id : S_p \to F$.

For each $x \in X$, let m'_x be the resulting bitwise mask of x, ie $m'_x = pha(I_x \cup id(S_x))$. Let $Y \subset X$, then $M(Y) = \sum_{x \in Y} m'_x$.

The assigment must be done in minimizing m'_x on all classes. This can be specified in several ways:

1. minimizing $M(X)$;
2. minimizing first m'_p, then $M(X)$;
3. X is partitioned in two subsets X_1 and X_2, with $p \in X_1$, and the optimization must first minimize $M(X_1)$, then $M(X_2)$.

The latter form is the more general.

Context X_1 is the set of classes that must be compiled now, whereas X_2 is the set of classes whose compilation is not yet required. They differ in the fact that each class in X_1 introduces a method that is multiply introduced and already called (ie a class site has been already compiled), whereas all the methods multiply introduced by classes in X_2 are not yet called.

Therefore, the optimization criterion can be understood as minimizing the hashtable size that are currently computed while minimizing the tables that will likely be computed later.

4.2.3 Subproblems

The problem can be simplified in two ways: either X or S_p can be singletons.

S_p is a singleton p introduces a single method that is already introduced in several classes, and the problem is to assign a free identifier to this method that minimizes the hash masks of all classes in X. This is a true instance of the problem considered, and it is certainly worthwhile to study this subproblem first. Although it would not be optimal, a heuristic solution to the general problem could be obtained with by iterating over S_p.

X is a singleton When X is a singleton, this is no longer an instance of the problem considered, since the point (4) in the problem input is no longer verified. Nevertheless, this is an instance of the *perfect numbering* problem [4], and the solution is as follows. The capacity of $m_p = pha(I_p)$ is defined as $2^{|b(m_p)|}$, and is the number of integers that m_p can discriminate. While the capacity is strictly less than the cardinality of $I_p \uplus S_p$, the least-weight 0-bit in m_p is switched to 1. This gives $m'_p = pna(I_p, |S_p|)$. A subset I'_p of F is then computed in such a way that $|\texttt{and}(m'_p, I'_p)| = |I'_p| = |S_p|$, and $\texttt{and}(m'_p, I'_p)$ and $\texttt{and}(m'_p, I_p)$ are disjointed. Free IDs in I'_p are then removed from F and successively assigned to symbols in S_p.

Perfect numbering can be simplified by considered that S_p is a singleton, too. Then $m'_p = pna(I_p, 1)$, and f is defined as the least element in F such that $\texttt{and}(m'_p, f) \notin \texttt{and}(m'_p, I_p)$. Iterating over S_p provides the same solution as above.

Perfect numbering is an essential step towards a solution of the problem considered.

4.3 Subproblem when S_p is a singleton

Subproblems may provide a better understanding of the general problem, along with a first, approximated solution.

4.3.1 Subproblem definition

The problem definition is simplified as follows.

Input

1. Y is a class set, with at least 2 members, and a distinguished element p;

2. each class $x \in Y$ is associated with a set I_x of integers;
3. a set of free identifiers $F \subset \mathbf{N} \setminus \bigcup_{x \in Y} I_x$.

Output the least free identifier $n \in F$ minimizing either $\sum_{x \in Y} pha(I_x \cup \{n\})$, or $pha(I_p \cup \{n\})$ then $\sum_{x \in Y} pha(I_x \cup \{n\})$.

4.3.2 Naive approach

A naive approach is an adaptation of perfect numbering to multiple masks, which would give the following algorithm.

Algorithm 1: Naive heuristics

```
Data: F, a free-identifier set; $s$, an overloaded method;
$Y$, a set of classes that know $s$, with masks $m_x = pna(I_x, 1)$ (all defined
as above)
Result: a free-identifier in F that can be assigned to $s$
free ← ⊥ ;
foreach $f \in$ F until free ≠ ⊥ do
    ok ← true ;
    foreach $x \in$ Y while ok do
        if and($f, m_x$) is not free for $x$ then
            ok ← false ;
    if ok then
        free ← $f$
return free
```

The algorithm enumerates free identifiers and checks whether they are hashed in free places in the hashtables of all concerned classes (Algorithm 1). m_x masks are assumed to be perfect hashing parameters for the method identifiers (I_x set) already known by x, with an extra free place, hence $m_x = pna(I_x, 1)$. Therefore, free places can be formally defined as $\mathtt{and}(m_x, \mathbf{N}) \setminus \mathtt{and}(m_x, I_x)$.

When Y is a singleton, this is exactly the perfect numbering algorithm, and there is always a solution if F is large enough. However, in the general case, the algorithm works when there is a solution with the current masks, but it does not terminate when the problem is over-constrained, ie when the free places of all concerned classes are not compatible with each other. Therefore, a realistic heuristics should scan only a finite set in order to detect that there is no solution, then some hashtables would be enlarged. This finite set might be a subset of F, or some set that would be representative of F.

4.3.3 Problem analysis and transformation

Bitwise and perfect hashing. Let x be a class in Y, and its associated hashtable, which is isomorphic to the $0..m_x$ range, has size $m_x + 1$ and can be analyzed in the following way. First $b(m_x) \subset \bigcup_{i \in I_x} b(i) = b(\bigvee_{i \in I_x} i)$ (or, equivalently, $m_x \in [\bigvee_{i \in I_x} i]$). Moreover, $K_x = \mathtt{and}(m_x, I_x)$ represents the set of occupied positions in the hashtable, and $\forall k \in K_x$, $k = \mathtt{and}(m_x, k)$. Thus, K_x can be considered as a kind of representative of I_x. The complement of K_x in the $0..m_x$ range is the set of unused entries, which can be partitioned into 2 subsets: U_x is the set of unreachable entries; A_x is the set of free entries that can be allocated for new identifiers. A position k in $0..m_x$ is said to be unreachable if $k \notin \texttt{and}(m_x, \mathbf{N})$. A necessary and sufficient condition is that $b(k) \backslash b(m_x)$ is not empty. In contrast, a position k is reachable if $b(k) \subset b(m_x)$, and a free position is a reachable position that is not occupied, hence not in K_x. Alternatively, $A_x = \texttt{and}(m_x, \mathbf{N}) \backslash K_x$. Overall, $0..m_x = K_x \uplus U_x \uplus A_x$. An essential (though trivial) property is that the number of reachable positions is a function of the 1-bit count of the mask, namely $|K_x \uplus A_x| = 2^{|b(m_x)|}$ [4].

From free positions to free identifiers. Part of the problem considered involves searching a free identifier that matches a free position in range $0..m_x$.

We could not imagine a constant-time technique that would return "the next free identifier that would fit the considered range". Instead, we propose sequential algorithms. In any case, the set F of free identifers can be implemented as a range union. Note that F is considered to be large enough to not be a problem constraint. Hereafter, we will make the informal assumption that, if there is an infinite series of numbers in $\mathbf{N}$ that satisfies a set of constraints, it is not disjointed from F.

The first technique consists in allocating an explicit hashtable of size $m_x + 1$, whose free entries are filled with a distinguished value. Then, for each free identifier $f \in F$, the entry at position $\texttt{and}(m_x, f)$ is checked. This is the technique used in our experiments of perfect numbering [4].

An alternative would involve enumerating $k \in A_x$ and checking if there is a free identifier $f \in F$ such that $\texttt{and}(m_x, f) = k$. However, we could not imagine a data structure that would allow us to implement this inverse problem.

From single class to multiple classes with mask-union. As mentioned above, the degenerated case where Y is a singleton represents the perfect-numbering problem, which is solved. In the problem considered here, the allocation of free identifiers must take into account several classes that introduce common methods.

Definition 4.1 *Assuming the previous definitions for Y, m_x and A_x, a number $k \in \mathbf{N}$ is said to be* PH-compatible *with Y iff $\forall x \in Y, \texttt{and}(m_x, k) \in A_x$.*

The key idea relies on the following observation. Let $x \in Y$ be a class, and m a mask such that $b(m_x) \subset b(m)$. Let $j \in 0..m$ be a position in m which is free for x, ie $\texttt{and}(j, m_x)$ is a free position for x. Then any free identifier $f \in F$ whose m-hashvalue is j will be free for x too, because of the bit-inclusion.

The idea can be generalized by considering the union of all masks implied, ie $m = \bigvee_{x \in Y} m_x = b^{-1}(\bigcup_{x \in Y} b(m_x))$. Then, the $0..m$ range is the finite set that must be considered. Indeed, if $k \in 0..m$ is PH-compatible with Y, then $f \in F$ will verify the same property if $\texttt{and}(m, f) = k$.

Proposition 4.2 *Let $m = \bigvee_{x \in Y} m_x$ be the bit-union of all considered masks. Let $f \in F$, and $k = \texttt{and}(m, f)$.*

Then k is PH-compatible *with Y iff f is too.*

Indeed, $\texttt{and}(k, m_x) = \texttt{and}(\texttt{and}(f, m), m_x) = \texttt{and}(f, \texttt{and}(m, m_x)) = \texttt{and}(f, m_x)$. $\square$

Algorithm 2: Realistic heuristics

```
Data: F, a free-identifier set; $s$, an overloaded method; Y, a set of
      classes that know $s$ (all defined as above)
Result: a free-identifier in F that can be assigned to $s$
/* initialization                                                  */
mask $\leftarrow 0$ ;
for $x \in$ Y do
    mask $\leftarrow$ or(mask, $m_x$)
frees $\leftarrow \emptyset$;
while frees $= \emptyset$ do
    foreach $j \subset$ mask do
        ok $\leftarrow$ true ;
        foreach $x \in$ Y while ok do
1           if and($j, m_x$) is not free for $x$;              /* failure */
            then
                ok $\leftarrow$ false ;
        if ok then
            frees $\leftarrow$ frees $\uplus j$
    if frees $= \emptyset$ then
        select $x \in$ Y and enlarge $m_x$ ;         /* enlarge some mask */
        update mask
foreach $f \in$ F do
    if and($f$, mask) $\in$ frees then
        return $f$
```

4.3.4 First Algorithm

Hence, a more realistic heuristics is Algorithm 2. This new algorithm is sound and terminates when F is reasonably large enough and the selection of the enlarged mask is appropriate.

For instance, a simple scheme involves selecting the class $x \in Y$ that maximizes the number of failures (line 1), and minimizes the number of free places, and enlargement consists of switching the least-weight 0-bit in m_x.

Furthermore, when no enlargement is needed, this algorithm is optimal for the first subproblem, in that it can enumerate any identifier free in all classes in Y. This is a direct consequence of the equivalence in Proposition 4.2.

However, the overall algorithm is not optimal in several ways:

- enlarging the mask m_x may lead to a solution at the next step, but enlarging another mask m_y might have led to a better solution;
- enlarging the mask m_x may be useless, because it does not lead to a solution at the next step and enlarging another mask m_y is required anyway; moreover enlarging only m_y would lead to a solution at the next step, but the selected identifier may occupy the enlarged m_x;
- as with perfect numbering, the selection of a free identifier is blindfold and never optimal for the future.

Algorithm 3: General algorithm

```
  Data: F, a free-identifier set; (X, ≺), a class hierarchy; p, a newly loaded
        class
  /* initialization                                                          */
  $p.mset \leftarrow$ the set of methods that are introduced in $p$ and were already
  introduced in some unrelated class, but have no identifier yet;
  $X_p \leftarrow$ the subset of classes in X that have methods in $p.mset$ ;
  Result: Assigns a free identifier to each method in $p.mset$
  foreach $x \in X_p$ do
      $x.mset \leftarrow$ the subset of methods in $p.mset$ known by $x$
      $x.iset \leftarrow$ the set of method identifiers known by $x$
1     $m_x \leftarrow pna(x.iset, |x.mset|)$;   /* perfect numbering parameter */
  foreach $s \in p.mset$ do
      /* subproblem                                                          */
      $Y_s \leftarrow$ the subset of classes in $X_p$ that have $s$;
      $Y \leftarrow \emptyset$ ;                          /* the $Y_s/\equiv$ quotient-set */
      mask $\leftarrow 0$ ;
      foreach $x \in Y_s$ do
2         if $\nexists y \in Y$, such that $x \equiv y$
          /* where $(x \equiv y) \iff (x.iset = y.iset \ \wedge \ m_x = m_y)$          */
          then
              $Y \leftarrow Y \uplus x$;
              mask $\leftarrow$ or(mask, $m_x$)
      frees $\leftarrow \emptyset$;
      while frees $= \emptyset$ do
          foreach $j \subset$ mask do
              $ok \leftarrow$ true ;
              foreach $x \in Y$ while $ok$ do
                  if and$(j, m_x)$ is not free for $x$;            /* failure */
                  then
                      $ok \leftarrow$ false ;
              if $ok$ then
                  frees $\leftarrow$ frees $\uplus j$
          if frees $= \emptyset$ then
3             select $x \in Y$ and enlarge $m_x$ ;        /* enlarge some mask */
              update mask
      foreach $f \in F$ do
          if and$(f$, mask$) \in$ frees then
              $s.id \leftarrow f$;
              $F \leftarrow F \backslash f$;
              foreach $x \in Y_s$ do
                  update $x$
              break
```

Algorithm 4: Mask enlargement

```
Data: mask and Y, as in Algorithm 3
Result: Switches the least-weight 0-bit in some masks in Y
failSet ← $\emptyset$;                /* a set of sets of classes in Y  */
minfailnb ← $maxInt$;                 /* common length of failSet members */
foreach $j \subset$ mask do
    fail($j$) ← $\emptyset$;
    foreach $x \in$ Y when and($j, m_x$) is not free for $x$ do
        fail($j$) ← fail($j$) $\uplus \{x\}$
    nb ← |fail($j$)|;
    if nb < minfailnb then
        minfailnb ← nb;
        failSet ← {fail($j$)}
    else if nb = minfailnb then
        failSet ← failSet $\uplus \{fail(j)\}$
bestFail ← heuristic selection in failSet;
foreach $x \in$ bestFail do
    switch least-weight 0-bit in $m_x$
```

4.3.5 Complete algorithm

We present now a complete algorithm which iterates over S_p and solves both difficulties (Algorithm 3). Let us consider first the initialization step. Each class x in X_p is initialized, with its sets of, respectively, method identifiers and new overloaded methods. Then, its mask m_x is initialized by perfect numbering (line 1), ie the mask computed from the identifier set, is enlarged in a minimal way in order to be able to contain the new identifiers that have to be computed. In parallel, the quotient set $Y/\equiv$ is computed, with the equivalence relationship defined by the equality of both masks and identifier sets (line 2). It is worth noting that the equivalence relationship depends upon the method s considered.

With Algorithm 3, another source of non-optimality is that methods in S_p are not ordered in an optimal way. For instance, the algorithm can run on a method s_1 which is under-constrained, and assign it a free place which would be essential for a method s_2 which is over-constrained, but will be handled later. Furthermore, Algorithm 2 is rather simple but can be markedly slow because the Y class-set can be very large (hundred, or even thousands, of classes). It is thus essential to find an equivalence relationship that allows us to quotient the class set. Enlarging masks can represent another bottleneck. Indeed, it may occur that several masks must be enlarged at the same time and iterating single-mask enlargement may represent a bad solution because the enlarged masks are not correlated if they do not fail at the same places.

The mask-enlargement algorithm, invoked line 3 in Algorithm 3, involves several steps (Algorithm 4): (i) determining the places j that minimize the failure number, ie the cardinality of the set $fail(j) = \{x \in Y \mid \texttt{and}(j, m_x) \text{ is not free for } x\}$; (ii) among the places j minimizing $|fail(j)|$, determining the place k such that $fail(k)$ minimizes some criterion, eg $\sum_{x \in fail(k)} m_x$; (iii) enlarging all the masks in $fail(k)$.

Table 1: Statistics of all methods

	introduced			defined			inherited			root
	total	μ	max	total	μ	max	total	μ	max	
visualworks2	17774	9.1	164	23738	12.1	179	608494	311.1	544	160
digitalk3	13004	9.6	440	17104	12.6	460	613996	452.8	1065	324
digitalk2	5534	10.4	271	6858	12.8	272	154796	289.9	677	182
IBM-SF	25000	2.8	257	116152	13.2	320	394375	44.9	346	5
JDK1.3.1	9567	1.3	149	28683	3.9	150	142445	19.2	243	5
Java.1.6	22098	4.4	286	35351	7.0	291	186061	36.7	669	11
JDK.1.0.2	3190	5.3	75	5095	8.4	78	22365	37.0	158	12
Self	26267	14.6	233	29415	16.3	233	1040415	577.4	969	1
Geode	8078	6.1	193	14214	10.8	207	305560	231.8	880	24
PRMcl	2369	4.9	115	3793	7.9	115	37517	78.3	208	29
Lov-obj-ed	3631	8.3	117	5026	11.5	127	37436	85.9	289	24
SmartEiffel	4854	12.2	222	7865	19.8	222	53704	135.3	324	1

The table presents, successively, the statistics of method introduction, method definition, and method inheritance, and each group depicts the total number on all classes and the average (μ) and maximum value per class. The last column represents the number of methods introduced in the hiearchy root.

4.4 General problem

Without loss of generality, the problem is not pairwise, but instead involves a graph (X_p, E) where two classes are related if they introduce common methods, ie E is formed of pairs (x, y) such that $S_x \cap S_y \neq \emptyset$. An alternative view is a bipartite graph (X_p, S_p, E'), where E' is formed of pairs (x, m) such that $m \in S_x$. The degree of each $m \in S_x$ is at least 2. In both graphs, p is connected to all X_p (resp. S_p).

An optimal solution to this problem remains an open issue. However, as our experiments show, in the next Section, that Algorithm 3 gives acceptable results, searching for an optimal solution would mostly be for the sake of it.

5 Experiments and evaluation

Perfect method hashing and numbering has been tested, with different variants, on a variety of benchmarks similar to those used in previous articles, eg [1, 16, 17]. All experiments are done with random class-loading, as in [4].

5.1 All-method perfect numbering

Table 1 presents the statistics on method definition in these benchmarks. The total of 'inherited methods' represents the memory occupation of method tables in the SST implementation (Figure 1), when it is relevant, or the positive part of method tables either in the MST implementation (Figure 2) or in the proposed implementation (Figure 4).

The next two tables present the results of perfect method numbering on these benchmarks, when classes are loaded with random leaf-class ordering. In Table 2(a) only methods are hashed, whereas classes are also hashed in Table 2(b). Therefore, the latter provides an implementation in multiple inheritance for both method invocation and subtype testing (Figure 3), and the latter must

Table 2: Statistics of PN for all methods in dynamic typing

(a) Leaf-class ordering, single inheritance (without class IDs)

275	optimal	useful PN-`and`			PN-`and`			SMI
20.6		min	μ	max	min	μ	max	
visualworks2	1.4	2.5	3.0	4.9	3.1	3.9	6.0	6.1
digitalk3	1.2	2.4	3.0	3.7	2.7	3.3	4.3	5.1
digitalk2	1.6	2.3	2.7	4.1	2.4	3.0	5.2	4.7
IBM-SF	1.4	3.6	4.5	7.4	6.3	8.8	11.6	5.2
JDK1.3.1	1.4	2.6	3.3	4.9	4.1	5.3	7.5	3.8
Java.1.6	1.4	2.7	3.5	5.2	4.0	5.5	7.9	3.6
JDK.1.0.2	1.5	2.0	2.4	3.3	2.5	3.2	4.5	3.4
Self	1.6	2.8	3.6	5.0	3.2	4.3	6.4	6.4
Geode	1.4	3.1	4.2	6.2	4.2	5.4	7.6	6.2
PRMcl	1.5	2.2	2.8	3.9	2.6	3.5	5.2	4.0
Lov-obj-ed	1.4	3.1	4.2	5.5	4.4	5.8	7.7	4.9
SmartEiffel	1.4	2.5	3.2	4.3	2.8	3.9	5.4	5.5

All numbers are normalized w.r.t. the SST method-table size. The first column presents the theoretical optimum based on the PH-mask 1-bit count [4]. The next group depicts the random statistics of the useful part of PH tables, ie the entries that are reachable according to the bit-wise mask. The last group depicts the random statistics of PH tables. For the sake of comparison, the last column depicts the method-table size with the subobject-based implementation.

(b) Leaf-class ordering, multiple inheritance (with class IDs)

275	optimal	useful PN-`and`			PN-`and`			SMI
20.6		min	μ	max	min	μ	max	
visualworks2	1.3	2.6	3.1	5.3	3.1	4.0	6.1	6.0
digitalk3	1.2	2.4	3.0	3.8	2.8	3.4	4.3	5.1
digitalk2	1.6	2.3	2.8	4.1	2.5	3.0	5.3	4.7
IBM-SF	1.4	3.5	4.7	6.5	7.5	9.5	13.7	5.1
JDK1.3.1	1.4	2.7	3.3	5.4	4.2	5.7	8.1	3.4
Java.1.6	1.5	3.0	3.7	5.3	4.8	6.2	8.1	3.4
JDK.1.0.2	1.4	2.0	2.4	3.4	2.6	3.3	5.3	3.3
Self	1.6	3.0	4.0	6.6	3.5	4.7	7.0	6.4
Geode	1.3	3.4	4.4	8.2	4.3	5.8	9.5	6.2
PRMcl	1.4	2.1	2.9	4.2	2.6	3.6	5.5	3.9
Lov-obj-ed	1.4	3.4	4.4	5.8	4.8	6.1	7.9	4.8
SmartEiffel	1.4	2.6	3.2	4.7	3.0	4.0	5.7	5.4

In this table, methods and classes are hashed in the same hashtable, as needed for the implementation in Figure 3.

(c) All-class ordering, single inheritance

40	optimal	useful PN-`and`			PN-`and`			SMI
127.1		min	μ	max	min	μ	max	
visualworks2	1.4	3.9	6.4	10.7	5.1	8.3	14.6	6.1
digitalk3	1.2	3.2	4.0	5.7	3.6	4.6	8.6	5.1
digitalk2	1.6	2.2	3.5	5.5	2.5	3.9	7.7	4.7
IBM-SF	1.4	6.0	7.6	10.1	10.3	15.2	52.7	5.2
JDK1.3.1	1.4	4.4	7.6	15.2	6.2	10.9	16.4	3.8
Java.1.6	1.4	4.3	6.7	11.0	6.4	10.6	29.9	3.6
JDK.1.0.2	1.5	2.1	2.8	4.9	2.8	3.7	5.9	3.4
Self	1.6	3.0	3.8	5.3	3.4	4.2	5.5	6.4
Geode	1.4	6.9	9.8	14.2	7.9	11.0	15.4	6.2
PRMcl	1.5	2.7	3.9	7.1	3.4	4.7	8.1	4.0
Lov-obj-ed	1.4	4.3	5.8	6.9	5.8	7.7	10.7	4.9
SmartEiffel	1.4	2.9	3.6	6.3	3.4	4.5	8.0	5.5

Table 3: Statistics of methods with single or multiple introduction

(a) Single introduction

	introduced			inherited		
	total	μ	max	total	μ	max
visualworks2	10465	5.4	153	506083	258.7	465
digitalk3	8577	6.3	326	559007	412.2	880
digitalk2	3902	7.3	215	139698	261.6	522

(b) Multiple introduction

	method	introduced			all inherited		shared inherited		
	number	total	μ	max	total	μ	total	μ	max
visualworks2	2112	6235	3.2	108	72226	36.9	52974	27.1	181
digitalk3	1481	4427	3.3	114	54989	40.6	41544	30.6	255
digitalk2	585	1632	3.1	75	15098	28.3	10765	20.2	158

The first column depicts the number of method selectors that are introduced in several classes. The next two column groups are similar to that of previous tables, and the last one presents the statistics of inherited methods when sharing is taken into account.

be used in single inheritance, when subtype testing is implemented as in SST. The last column represents the method-table size in the C++-like subobject-based implementation of, multiple implementation (SMI), which can be considered as a negative reference as it is, in the worst-case, cubic in the number of classes, instead of quadratic as in most implementations [17]. All numbers are expressed as ratios to the SST implementation, ie the 'total inherited' column in Figure 1. Apart from the aforementioned impossibility to optimize this use of perfect hashing in an adaptive compiler, these results show that the memory requirement of this implementation is markedly higher than the SST implementation. The ratio to SST is more than 2 on average, and not markedly higher than the theoretical optimal (which is, however, not reachable, see [4]). It can exceed 7 on some benchmarks in the worst cases. Actually, in these worst cases (column max), it is not better than SMI.

Finally, Table 2(c) presents the same statistics as Table 2(a), but classes are now loaded with random all-class ordering. Interested readers are referred to [4] for an in-depth discussion about leaf-class vs all-class orders. Roughly speaking, the set of leaf-classes is close to the set of actual concrete classes, whose instantiation triggers class loading. Therefore, considering only leaf-class orders is closer to the expected behaviour of real programs, and PH/PN yields far better results with leaf-class orders.

5.2 Perfect numbering restricted to overloaded methods

The second experiment concerns class hierarchies in single inheritance, when methods are distinguished from each other according to the number of their introduction classes. The considered benchmarks are versions of SMALLTALK. Table 3 presents the statistics of method definition according to whether methods are introduced by a single class or several ones. They show that most methods are introduced by a single class. Hence, perfect hashing could considered because it would apply to a small subset of all method invocations, and the required memory requirement would remain low.

Table 4: Statistics of PN for overloaded methods, CWA heuristics

(a) Unshared

130	hashed	optimal	useful PN-`and`			PN-`and`			SST
43.6			min	μ	max	min	μ	max	
visualworks2	36.9	55.2	124.1	197.2	615.2	209.6	371.6	1099.2	295.7
digitalk3	40.6	58.7	122.1	184.0	289.1	176.0	279.3	798.9	452.8
digitalk2	28.3	38.2	63.3	88.4	136.4	83.5	130.9	402.3	289.9
visualworks2	36.9	55.2	92.7	107.2	138.5	144.7	176.3	280.2	295.7
digitalk3	40.6	58.7	106.1	128.5	149.0	147.4	183.6	308.2	452.8
digitalk2	28.3	38.2	55.4	69.1	98.3	74.5	93.6	157.4	289.9

(b) Shared

130	hashed	optimal	useful PN-`and`			PN-`and`			SST
43.6			min	μ	max	min	μ	max	
visualworks2	27.1	39.6	100.6	149.9	378.7	172.7	272.7	680.1	295.7
digitalk3	30.6	44.3	99.0	150.5	222.4	146.4	214.3	466.1	452.8
digitalk2	20.2	27.6	49.3	66.2	93.2	66.0	95.0	237.2	289.9
visualworks2	27.1	39.6	75.0	86.3	114.2	122.2	147.9	226.2	295.7
digitalk3	30.6	44.3	88.5	108.1	127.0	124.0	153.4	223.3	452.8
digitalk2	20.2	27.6	43.3	54.4	73.6	59.8	74.9	104.6	289.9

Each column presents the average number per class of, successively, the theoretical PH optimum, the observed random statistics of useful PH entries, and all PH entries. For the sake of comparison, the last column depicts the number of entries in the SST method table.
Each table is split into two subtables: the upper one presents the statistics on all class-loading orders, whereas the lower one restricts the statistics on leaf-class loading orders that are presumed to be closer to actual concrete-class loading.
The total extra size for the PH approach must be increased by two, for taking into account the extra indirection and hash parameter in the method table.

Finally, Table 4 presents the statistics of PN when it is restricted to overloaded methods. Benchmarks are a few SMALLTALK class hierarchies. This approach is, however, not realistic in that it supposes that overloaded methods are identified under the closed world assumption. It is presented only in order to examine the efficiency of PN.

Results are now presented as an average size per class, instead of being a ratio to SST, as in Table 2. The last column recalls the SST average (column 'inherited μ' of Table 1). On average, the PH memory requirement is now lower than for SST and the overall worst-case ratio remains lower than 3.5. The space-efficiency is thus better than for all-method PH, and the time efficiency is even better since most PH invocations can be shortcut. However, in itself, the algorithmic efficiency is far lower. Indeed, in the worst cases of Table 2(a), all ratios are less than fivefold the optimal. In contrast, in Table 4(a), the worst cases are almost 20 times the optimal. This is simply explained by the fact that the optimal is reached with a single-inheritance hierarchy, when all methods are introduced by a single class [4]. Hence, methods with multiple introduction represent bad cases, and this approach considers only bad cases.

Table 5: Statistics of PH for overloaded methods with Algorithm 3

(a) Unshared

6545 0.0	hashed	optimal	useful PN-`and` min	 μ	 max	PN-`and` min	 μ	 max	SST
visualworks2	36.9	55.2	75.6	96.2	190.4	103.8	164.2	1625.7	295.7
digitalk3	40.6	58.7	82.8	103.2	188.1	101.9	137.6	353.5	452.8
digitalk2	28.3	38.2	46.7	55.4	75.0	53.6	69.8	218.7	289.9
visualworks2	36.9	55.2	71.8	79.4	96.6	90.3	108.5	137.2	295.7
digitalk3	40.6	58.7	78.1	89.8	107.2	94.6	117.8	177.1	452.8
digitalk2	28.3	38.2	45.7	52.6	66.6	51.6	63.3	81.9	289.9

(b) Shared

6545 0.0	hashed	optimal	useful PN-`and` min	 μ	 max	PN-`and` min	 μ	 max	SST
visualworks2	27.1	39.6	57.9	76.3	128.7	81.9	125.6	930.4	295.7
digitalk3	30.6	44.3	63.5	81.7	165.1	79.1	106.8	215.5	452.8
digitalk2	20.2	27.6	34.8	41.3	56.8	39.4	52.3	129.9	289.9
visualworks2	27.1	39.6	53.9	60.9	77.7	69.1	85.5	109.1	295.7
digitalk3	30.6	44.3	59.3	69.4	84.7	74.1	91.5	130.3	452.8
digitalk2	20.2	27.6	34.0	39.2	50.0	38.9	48.3	64.6	289.9

Same content as in Table 4, with a different algorithm.

5.3 Experiments of Algorithm 3

Finally, we experimented Algorithm 3 on a few SMALLTALK class hierarchies, with random class-loading, as in [4].

Space efficiency. Table 5 sums up our results with respect to the memory occupation required by the hashtables. Each subtable concerns, respectively, the cases where sharing is, or is not, taken into account. A subclass may share the hashtable of its direct superclass, when the subclass does not introduce any proper overloaded method. Moreover, each subtable is in turn split into two subtables, for all-class and leaf-class orders, respectively.

For instance, the results for the visualworks2 benchmark, in the shared case with leaf-class loading (ie the lowest subtable in Table 5), can be read as follows. On average there are 27.1 methods per class that must be hashed, and it sets the average mask lower-bound to 39.6, based on the number of 1-bits required for hashing the considered methods. The ratio between these two numbers is always in range 1..2, and it represents the optimal occupation ratio (Proposition 3.7 in [4]). The PN-`and` columns present the statistics of the hashtable size according to class loading orders. On average, it is about twofold the optimal and 3-fold the method number, and the deviation is rather small (about 10-20%). The useful PN-`and` columns presents the same statistics, restricted to the hashtable entries that are actually reachable. The difference could be reused for allocating other data. Finally, for the sake of comparison, the last column presents the average size of the method tables, based on the SST implementation. It shows that hashtables would represent only 30% of the method tables. Although not negligible, it remains quite reasonable in comparison with all-method PH or C++-like subobject-based implementation, both techniques whose ratio to SST

Table 6: Statistics on iterations in Algorithm 3

(a) Iterations

	method number	iteration number 0	1	2	3	4	5	over 6	max
visualworks2	2112	94.1	4.3	1.0	0.4	0.2	0.1	0.0	10
digitalk3	1481	94.2	4.2	0.9	0.4	0.2	0.1	0.0	10
digitalk2	585	91.5	5.6	1.7	0.8	0.3	0.1	0.0	10
visualworks2	2112	92.3	5.3	1.4	0.6	0.3	0.1	0.0	11
digitalk3	1481	91.0	5.9	1.7	0.8	0.4	0.1	0.0	11
digitalk2	585	89.2	7.0	2.2	1.1	0.4	0.1	0.0	11

(b) Fail number

	reiteration number	min failure number 1	2	3	4	5	6-10	11-20	over 21	max
visualworks2	178	67.0	14.0	6.7	4.0	2.5	4.4	1.2	0.1	39
digitalk3	126	68.1	13.8	6.0	3.3	2.0	4.2	2.6	0.0	24
digitalk2	77	64.9	14.6	6.3	4.0	3.3	6.8	0.2	0.0	22
visualworks2	251	58.1	14.8	7.6	4.8	3.4	7.6	3.2	0.5	50
digitalk3	208	58.4	17.1	7.6	4.0	2.5	6.4	3.7	0.1	39
digitalk2	100	59.8	15.3	7.4	4.7	3.6	8.3	0.8	0.0	25

The first subtable depicts the statistics on the number of iterations required for hashing each overloaded method, and the first column is the number of overloaded methods.
The second subtable depicts the statistics on the minimal number of failures, ie the number j that minimizes $|fail(j)|$, at each iteration of the algorithm. The first column is the average reiteration number.
Each column j represents the percentage of computations that, respectively, required j iterations, or presented at least j failures. The last column represents the maximum observed value for j.

can exceed 6 (Table 2).

Runtime-efficiency of the algorithm. Regarding the runtime-efficiency of the algorithm, it appears that it is not that fast, and markedly slower than our previous applications of perfect hashing. Therefore, a careful implementation seems to be essential.

Table 6 presents some statistics about the behaviour of the algorithm which is rather satisfactory. Table 6(a) shows that there are few (less than 10%) enlargement cases (ie line 3 in Algorithm 3) and, most of the times, a single iteration is sufficient. In very exceptional cases (less than 1%%), more than 5 iterations are required. The overall number of reiterations is about 12-18% of the method number. Table 6(b) presents statistics on the minimum failure number, ie the minimum number of masks that must be enlarged, for each iteration. In contrast with the iteration number which is rather low, when the current masks are not compatible with an extra identifier, the minimum number of failures is 1 in more than 50% of cases, but it may be greater than 20 in a very few cases.

Recomputation cost. The recomputation cost is less easy to assess, because our simulation is eager, and hashtable-recomputation count is markedly exaggerated in comparison with a lazy behaviour. Anyway, Table 7 presents statistics on different parameters. The first column group represents the count of all

Table 7: Statistics of recompilation for Algorithm 3

6545	all shared			shared enlarged			classes		
	update number			allocation number			HT		
0.0	min	μ	max	min	μ	max	number		all
visualworks2	4137.	5156.	6693.	1419.	1570.	1817.	1066	1938	1956
digitalk3	3182.	3870.	4859.	995.	1131.	1421.	753	1330	1356
digitalk2	970.	1230.	1632.	401.	481.	604.	301	526	534
visualworks2	5739.	6697.	8076.	1576.	1796.	2081.	1066	1938	1956
digitalk3	4162.	4938.	5739.	1116.	1317.	1603.	753	1330	1356
digitalk2	1191.	1446.	1675.	470.	548.	636.	301	526	534

hashtable computation and recomputations, when sharing is taken into account, but not the fact that the hashtable size may remain unchanged in the computation. The second group represents the count of recomputations that involve an enlargment of the considered hashtables. Thus it is the exact number of hashtables that would be allocated during the loading of all classes. The third group presents different class numbers: classes that (i) introduce or (ii) know overloaded methods, then (iii) all classes. The former is exactly the number of shared non-empty hashtables that are required when all of the classes have been loaded. In contrast, the number of shared recomputation represent the number of dynamic allocations of an hashtable which is larger than the previous one. Implicitly, each class is initialized with a common empty hashtable, ie a hashtable with a single empty entry. Therefore, the test shows that the number of allocations is higher than what is required, in a ratio less than 2 on average. Finally the number of all shared recomputations is far higher, but they mostly involve updating an existing hashtable which does not require to be enlarged, and the extra cost only implies a few assignments.

For instance, with the visualworks2 benchmark there are 1066 classes introducing overloaded methods. On average, with leaf-class ordering (lower subtable), 6697 hashtable updates are needed, including 1796 allocations.

Leaf-class vs all-class orders. Finally, it is also worth comparing the results and behaviour of Algorithm 3 with respect to leaf-class vs all-class orderings, although they represent an alternative in the simulation, not in an actual execution which should be midway between.

Leaf-classes provide far better results in terms of hashtable size (Table 5), but they are indeed more costly in terms of iterations and failures (Table 6), and for recomputations (Table 7).

6 Related Works, Conclusions and Prospects

6.1 Related works

SMALLTALK and SELF have been pioneers of object-oriented implementation and compilation in a dynamic typing and dynamic loading setting. In this context, all the techniques that we are aware of rely on *inline caches* [18, 19, 12]. An inline cache can be viewed as a guarded monomorphic call. The receiver's dynamic type is compared to an expected type, and a success yields a static call. In case of failure, an obscure process called *lookup* is performed, and looks up in the class hierarchy for the method that must be invoked.

There are a variety of inline caches, and they can be static or dynamic, mono- or poly-morphic. While a static cache[5] is immutable and results from static compilation, dynamic caches are mutable and result from the runtime behaviour of the program; for instance, a cache miss yields the update of the cache with the lookup result; runtime profiling is also possible. Polymorphic caches involve more than one expected type. Overall, inline caches present pros and cons. When their guard succeeds, they are very efficient because of conditional-branching prediction of modern universal processors. Hence, in the best cases, inline caches are almost as efficient as static calls. These best cases include the cases where the receiver is monomorphic (see footnote 2). When the invocation, instead of the receiver, is monomorphic, the guard must access the method table in order to compare the invoked method with the expected one. This is mainly used for inlining.

What about bad cases? In practice, the failure case, ie the famous lookup, is inefficient. Indeed, we are not aware of any constant-time technique available in dynamic typing apart from perfect hashing. Moreover, a cache might be efficient at a moment, not during the whole programm execution. For instance, in the program prologue, an invocation site may have receivers of type A, then receivers of type B during the rest of the execution. If the site is optimized according to the prolog it will be unoptimized for the main part of the execution. Polymorphic caches are a solution, but it is always possible to build a bad case from any good situation. Another solution involves dynamic caches, but runtime profiling is so costly that the solution might be worst than the problem. Inline caches also yield longer code sequence, which increase the overall code size. While the thunk approach is still possible, it cannot be envisaged to factorize the same thunk between several similar invocation sites because the prediction of conditional branching would lose its accuracy.

In contrast, our proposal does not involve any of the drawbacks of inline caches, and it provides better solution for some of their best aspects, eg for monomorphic invocations instead of monomorphic receivers. The worst case of our proposal, ie perfect hashing, is likely more efficient than the lookup, although the latter might use the former. Furthermore, it remains possible to couple inline caches with the less efficient sequences generated according to our proposal, which are not so many.

Distinguishing between overloaded and non-overloaded methods was also proposed in [20], however in the context of global, static compilation.

6.2 Conclusion and prospects

In this paper, we proposed a novel object representation for method dispatch in dynamic typing, single inheritance and dynamic loading. To our knowledge, this is the first constant-time implementation of method dispatch in this context that requires reasonable memory occupation.

This technique involves hashing overloaded methods, and its efficiency relies on the fact that they are not too many.

[5]A static cache looks like an oxymoron.

Simulation over a few Smalltalk benchmarks show that the technique is promising. Indeed, the number of overloaded methods is actually low, and the overall hashtable size remains reasonable. However, the simulation we carried out is unable to assess the recompilation cost, because it is markedly more eager than would be an actual compiler. Hence the cost that we observed are exaggerated.

Therefore, the main prospects of this work is to perform simulations that would be closer to actual executions, for instance by adpating to dynamic typing the simulations used in [7]. It would be worth considering, too, the possibility of simulating these implementation and recompilation protocol in a Smalltalk virtual machine via meta-programming.

References

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

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

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

[4] R. Ducournau and F. Morandat. Perfect class hashing and numbering for object-oriented implementation. *Softw. Pract. Exper.*, 41(6):661–694, 2011.

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

[6] F. Morandat and R. Ducournau. Empirical assessment of C++-like implementations for multiple inheritance. In *Proc. ICOOOLPS Workshop*, pages 7–11. ACM, 2010.

[7] R. Ducournau and F. Morandat. Towards a full multiple-inheritance virtual machine. *Journal of Object Technology*, 12:29, 2012.

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

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

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

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

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

[13] R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. *Science of Computer Programming*, 76(7):555–586, 2011.

[14] D.F. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*, SIGPLAN Not. 31(10), pages 324–341. ACM, 1996.

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

[16] R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. *Softw. Pract. Exper.*, 41(6):627–659, 2011.

[17] R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comp. Surv.*, 43(4), 2011.

[18] T. J. Conroy and E. Pelegri-Llopart. An assessment of method-lookup caches for Smalltalk-80. In Krasner, editor, *Smalltalk-80 Bits of History, Words of Advice*, pages 238–247. 1983.

[19] L. P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. ACM Symp. on Principles of Prog. Lang. (POPL'84)*, pages 297–302, 1984.

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