Perfect Method Hashing in a Dynamic Setting

Roland Ducournau
LIRMM – Université Montpellier 2, France
ducournau@lirmm.fr

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.

Categories and Subject Descriptors

D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages,* SMALLTALK; D.3.3 [**Programming languages**]: Language Constructs and Features—*classes and objects, inheritance*; D.3.4 [**Programming languages**]: Processors—*compilers, run-time environments*; E.2 [**Data**]: Data Storage Representations—*object representations*

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

adaptive compiler, closed-world assumption, compilation protocol, dynamic loading, late binding, method tables, multiple inheritance, open-world assumption, perfect hashing, subtype test, virtual machine

ICOOOLPS'12, June 11, 2012, Beijing, China

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* [Ducournau, 2008] and based on *perfect hashing*, ie truly constant-time, collision-free hashing [Sprugnoli, 1977, Czech et al., 1997]. From an algorithmic point of view, a variant called *perfect numbering* involves optimizing the class IDs in order to minimize the hashtable sizes [Ducournau and Morandat, 2011]. Real-size experiments in the PRM compiler [Ducournau et al., 2009] 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 [Morandat and Ducournau, 2010]. Therefore, we also proposed a compilation/recompilation protocol that would allow for an efficient implementation in a *just-in-time*, dynamic compiler [Ducournau and Morandat, 2012]. 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 [Goldberg and Robson, 1983], 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

[1] This use of the *overload* term must not be confused with

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 [Ducournau, 2008].

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 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 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 satisfies 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* [Cohen, 1991], with the class ID in each method group. However, the SST approach cannot be used either with multiple inheritance or dynamic typing.

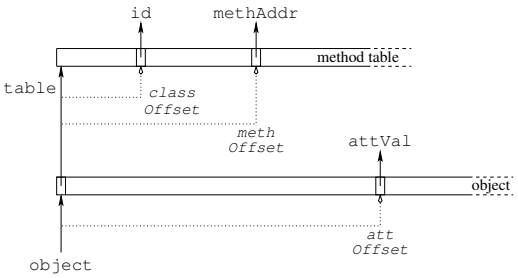
Perfect class hashing in static typing. Figure 2 describes the implementation of method invocation and subtype testing with perfect class hashing, in a static typing setting. It can be used directly for multiple subtyping (ie JAVA interfaces). In the method tables, the positive offsets contain

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, at run-time, on all their parameters [Castagna, 1997]. Finally, it must not be confused with *overriding*, which denotes the fact that a method defined in a class can be redefined in a subclass.

```
// 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 [Driesen, 2001]. Pointers and pointed values are in Roman type with solid lines, and offsets are italicized with dotted lines.

Figure 1: Single-subtyping implementation

the SST implementation which is used for class-typed invocations. Interface IDs are hashed and method addresses retrieved in the group introduced by the interface that introduces the considered method. The technique can also be extended to attribute access for full multiple inheritance, but this extension (called accessor simulation) is not that efficient. In [Ducournau and Morandat, 2012], 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. Furthermore, 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.

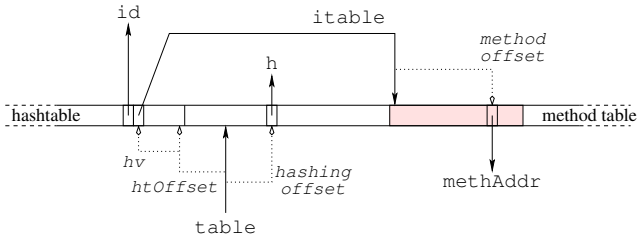
All-method perfect hashing in dynamic typing. Methods can be hashed directly, too. Figure 3 describes the implementation of method invocation and subtype testing in dynamic typing, with perfect method hashing. In single inheritance, it would be more efficient to remove class IDs from the hashtable and 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

[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).

```
//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

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.

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

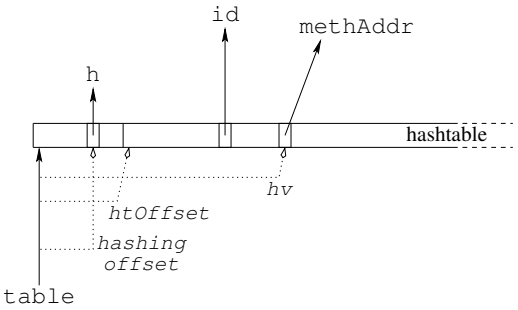
Besides this general object representation, dynamic typing yields a few specific cases which complicate the generated code and add overhead.

- The unknown-method exception is a consequence of runtime type errors, and each method invocation must check that the right method is invoked; this is simply done by comparing the invoked selector ID with the expected one in the method preamble.

- Primitive values encode their type as a tag in their bitwise representation; they are thus accessed via specific techniques like *polymorphic inline cache* (PIC) [Hölzle et al., 1991].

- A `null` value is used for uninitialized variables and fields; it can be considered as a special type tag.

```
//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

Generally, these specific cases must be combined, since a given invocation site might yield each of them at runtime. Hence, it markedly increase the code sequence length. However, some of these points are irrelevant in a few specific cases, eg when the receiver is `self`, or for those methods that are introduced by the hierarchy root. Anyway, the technique proposed here does not address theses special cases, which would be tackled as usual.

3. COMPILATION PROTOCOL

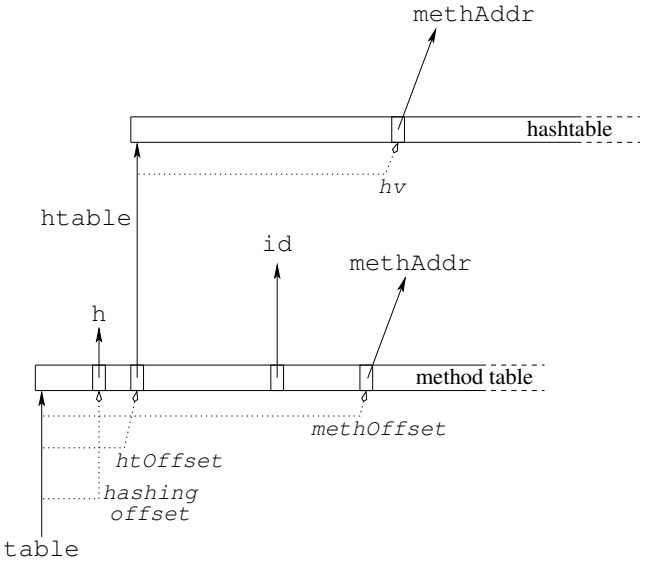
As explained in the introduction, we are concerned, here, with dynamic compilation. The compilation/recompilation protocol is in charge of a few tasks: (i) computing the data structures associated with a class; (ii) generating machine-code from method source code (or bytecode); (iii) recompiling some 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.

```
// 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

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 [Ducournau and Privat, 2011]). 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 [Ducournau and Privat, 2011]. 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 selector 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) [Bacon and Sweeney, 1996], 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) [Dean et al., 1995]. 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 a companion technical report [Ducournau, 2012].

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.

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.

The complete combination gives the decision tree in Figure 5.

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 (*code patching*) in order to call it. A mixed approach should likely be preferred. There are well-known issues, when the recompiled method is currently active, or for register allocation with code patching, but they remain 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

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 → PH implementation
      ii. *the introduction subclasses are only primitive types* → PIC implementation
      iii. otherwise → combined PIC-PH implementation
3. otherwise
   (a) *the method is unknown* → 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) → static call;
      ii. it is not redefined in primitive types → SST implementation;
      iii. otherwise → PIC+SST implementation;
   (c) *the method is introduced by a single class* D ⇒ (2-c);
   (d) otherwise ⇒ (2-d);

Conditions are tested sequentially (ie each condition implies the negation of the previous ones). Italic type denotes provisory conditions.

Figure 5: Decision tree of the compilation protocol

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 [Ducournau, 2012].

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 numbers of hashtable recomputations and allocations, and method recompilations; (iii) the time-efficiency of the generated code, which depends on the numbers of sites according to their respective implementation. In the following, we will consider only the first two criteria, hashtable size and recomputation number. When a newly overloaded method is hashed, the hashtables of all of the living classes that have this method must be recomputed. For each class, there are two cases: (a) the new hash parameter is the same as previously, and the hashtable has just to be updated by assigning the new method at curretnly free palces; (b) the hashtable must be enlarged, by allocating a new hashtable and reinitializing it. Only the last case presents a significant cost.

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 a direct 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.

- The hashtable computation and allocation 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 will likely increase the number of recomputations, each time a new overloaded method becomes alive. Our experiments are too coarse-grained to take method liveness into account.

4. EXPERIMENTS AND EVALUATION

Perfect method hashing and numbering has been tested, with different variants, on a few SMALLTALK benchmarks similar to those used in previous articles, eg [Ducournau, 2008, 2011]. All experiments are done with random class-loading, as in [Ducournau and Morandat, 2011].

4.1 All-method perfect numbering

Table 1: Statistics of method selectors 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

The table depicts the number of selectors that are introduced in a single class, along with the average and maximum numbers per class. The second group depicts the number of corresponding methods known by all classes.

(b) Multiple introduction

	method number	introduced		
		total	μ	max
visualworks2	2112	6235	3.2	108
digitalk3	1481	4427	3.3	114
digitalk2	585	1632	3.1	75

The table depicts the number of method selectors that are introduced in several classes, then the statistics per class.

(c) Taking sharing into account

	all inherited		shared inherited		
	total	μ	total	μ	max
visualworks2	72226	36.9	52974	27.1	181
digitalk3	54989	40.6	41544	30.6	255
digitalk2	15098	28.3	10765	20.2	158

The two column groups present the number of inherited overloaded methods per class, and they differ by the fact that sharing is taken into account or not.

We first tested all-method perfect numbering, which appears to be rather space inefficient. The place is missing, here, for reporting these tests and readers are referred to the companion technical report [Ducournau, 2012].

4.2 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. Table 1 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 be considered because it would apply to a small subset of all method invocations, and the required memory requirement would remain low.

Finally, we experimented the optimized algorithm of [Ducournau, 2012] on these SMALLTALK class hierarchies, with random leaf-class loading, as in [Ducournau and Morandat, 2011].

Results are now presented as an average size per class, instead of being a ratio to SST, as in Table **??**. The last column recalls the SST average (column 'inherited μ' of Table **??**).

Table 2: Statistics of PH for overloaded methods

	hashed	optim.	PN-`and`			SST
			min	μ	max	
visualworks2	27.1	39.6	69.1	85.5	109.1	295.7
digitalk3	30.6	44.3	74.1	91.5	130.3	452.8
digitalk2	20.2	27.6	38.9	48.3	64.6	289.9

Each column presents the average number per class of, successively, the hashed methods, the theoretical PH optimum, the observed random statistics of PH entries. For the sake of comparison, the last column depicts the number of entries in the SST method table. 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.

Table 3: Statistics of recompilation

	all updates		shared allocations		number of	
	min	max	min	max	HT	classes
visualworks2	5739.	8076.	1576.	2081.	1066	1956
digitalk3	4162.	5739.	1116.	1603.	753	1356
digitalk2	1191.	1675.	470.	636.	301	534

Space efficiency. Table 2 sums up our results with respect to the memory occupation required by the hashtables. Sharing is taken into account, as a subclass may share the hashtable of its direct superclass, when the subclass does not introduce any proper overloaded method.

For instance, the results for the visualworks2 benchmark 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 [Ducournau and Morandat, 2011]). 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 can exceed 6 (Table **??**). Moreover, it would replace the data structures, eg hashtables, used for the usual Smalltalk lookup.

One might observe that the results of perfect class numbering are markedly closer to the optimum than these. This is simply explained by the fact that the optimum of perfect class hashing is reached with a single-inheritance hierarchy [Ducournau and Morandat, 2011], and this extends to method hashing when all methods are introduced by a single class, too. Hence, methods with multiple introduction represent bad cases, and this approach considers only bad cases.

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.

Recomputation cost. The recomputation cost is less easy to assess, because our simulation is eager, and the hashtable-recomputation count is markedly exaggerated in comparison with a lazy behaviour. Anyway, Table 3 presents statistics on different parameters. The first column group represents the count of all hashtable computation and recomputations, when sharing is taken into account. However, the fact that the hashtable size may remain unchanged in the computation, hence making the update straitghtforward, is not considered. In contrast, 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 a hashtable which is larger than the previous one. Initially, 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.

5. RELATED WORKS

Smalltalk and Self have been pionneers 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* [Conroy and Pelegri-Llopart, 1983, Deutsch and Schiffman, 1984, Hölzle et al., 1991]. 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[3] 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 pro-

[3] A static cache looks like an oxymoron.

cessors. 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. Therefore, to our knowledge, there is currently no efficient solution to method dispatch for highly polymorphic invocation sites, whose receiver's dynamic type is constantly changing and the invoked method covers a large set of several tens methods. In these situations, the proposed approach is a marked improvement.

Moreover, when the receiver's dynamic type is steadier, the 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 prologue 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 increases the overall code size. While it could be envisaged to factorize the same polymorphic cache between several similar invocation sites, experiments shows that the prediction of conditional branching loses 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 [Vitek and Horspool, 1994], however in the context of global, static compilation.

6. CONCLUSION AND PROSPECTS

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

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

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 adapting the simulations used in [Ducournau and Morandat, 2012] to dynamic typing. It would be worth considering, too, the possibility of simulating these implementation and recompilation protocol in a SMALLTALK virtual machine via meta-programming.

Finally, this proposal is mostly cross-cutting standard optimizations like method inlining, and we do not expect that it will increase or decrease the difficulties raised by adaptive optimizations. However, real-size experiments are needed to confirm this expectation.

References

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.

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

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

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.

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

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.

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.

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

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

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

R. Ducournau. Perfect hashing for method dispatch with dynamic typing and dynamic compilation. Research Report 12-010, LIRMM, Université Montpellier 2, 2012.

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

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

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

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.

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

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.

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

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

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.