Empirical Assessment of Object-Oriented Implementations with Multiple Inheritance and Static Typing

Roland Ducournau Floréal Morandat
LIRMM – Université Montpellier 2, France
{ducournau,morandat}@lirmm.fr

Jean Privat
Université du Québec à Montréal, Canada
privat.jean@uqam.ca

Abstract

Object-oriented languages involve a trade-off between three aspects, namely multiple inheritance, runtime efficiency and open world assumption (OWA), i.e. dynamic loading. The runtime efficiency of object-oriented programs is conditioned by the underlying *implementation technique* and *compilation scheme*. The former is concerned by the precise data structures that support basic object-oriented mechanisms (namely method invocation, attribute access and subtype testing). The latter consists of the production line of an executable program from the source code files, including compilers, linkers, loaders, virtual machines and so on. Many implementation techniques have been proposed and several compilation schemes can be considered from fully global compilation, under the closed-world assumption, to fully separate compilation, with dynamic loading, under the OWA, with midway solutions that involve separate compilation and global linking. In this article, we review a significant subset of all possible combinations and present a systematic empirical comparison of their respective efficiency with *all other things being equal*. The testbed consists of the PRM compiler that has been designed to implement various alternative techniques, in different compilation schemes. The considered techniques include C++ subobjects, coloring, perfect hashing and binary tree dispatch. A variety of processors have been considered. Qualitatively, these first results confirm the intuitive or theoretical abstract assessments of the tested approaches—as expected, efficiency increases as CWA strengthens. From a quantitative standpoint, the results are the first to precisely compare the efficiency of techniques that are closely associated with languages, e.g. C++ and EIFFEL. They also confirm that perfect hashing should be used for implementing JAVA interfaces.

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

General Terms Experimentation, Languages, Measurement, Performance

Keywords binary tree dispatch, close world assumption, coloring, downcast, dynamic loading, interfaces, late binding, method tables, multiple inheritance, multiple subtyping, open world assumption, perfect hashing, single inheritance, subtype test, type analysis, virtual function tables

1. Introduction

In spite of its 30-year old maturity, object-oriented programming still presents an important efficiency issue in the context of *multiple inheritance* and this issue is worsened by *dynamic loading*. In a recent article (Ducournau 2008), we identified three requirements that all implementations of object-oriented languages, especially in this context, should fulfil—namely (i) *constant-time*, (ii) *linear-space* and (iii) *inlining*. This implementation issue is exemplified by the two most used languages that support both features, namely C++ and JAVA. When the `virtual` keyword is used for inheritance, C++ provides a fully reusable implementation, based on subobjects, which however implies a lot of compiler-generated fields in the object layout and pointer adjustments at run-time[1]. Moreover, it does not meet the linear-space requirement and there is no known efficient subtype test available for this implementation. JAVA provides multiple inheritance of interfaces only but, even in this restricted setting, the current implementations are not constant-time (see for instance (Alpern et al. 2001a)). The

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

present research was motivated by this observation—though object-oriented technology is mature, the ever-increasing size of object-oriented class libraries makes the need for scalable implementations urgent and there is still considerable doubt over the scalability of existing implementations.

The implementation of object-oriented languages relies upon three specific mechanisms, namely *method invocation*, *subtype testing* and *attribute access*. Method invocation implies *late binding*—that is, the address of the actually called procedure is not statically determined at compile-time, but depends on the dynamic type of a distinguished parameter known as the *receiver*. Subtyping and inheritance introduce another original feature, i.e. run-time subtype checks, which amounts to testing whether the value of x is an instance of some class C or, equivalently, whether the dynamic type of x is a subtype of C. This is the basis for so-called *downcast* operators. An issue similar to late binding arises with attributes (aka *fields*, *instance variables*, *slots*, *data members* according to the languages), since their position in the object layout may depend on the object's dynamic type.

Message sending, attribute access and subtype testing need specific implementations, data structures and algorithms. In statically typed languages, late binding is usually implemented with tables, called *virtual function tables* in C++ jargon. These tables reduce method calls to function calls, through a small fixed number—usually 2—of extra indirections. It follows that object-oriented programming yields some overhead, as compared to usual procedural languages. When static typing is combined with single inheritance—this is *single subtyping*—two major invariants hold: (i) a *reference* to an object does not depend on the static type of the reference; (ii) the *position* of attributes and methods in the tables does not depend on the dynamic type of the object. These invariants allow direct access to the desired data and optimize the implementation. Hence, all three mechanisms are time-constant and their constant is small and optimal. The code sequence is short and easily inlinable. Finally, the overall memory occupation is linear in the size of the specialization relationship—this can be understood as a consequence of the fact that constant-time mechanisms require some compilation of inheritance. Otherwise, dynamic typing or multiple inheritance make it harder to retain these two invariants.

Implementation is thus not a problem with single-subtyping languages. However, there are almost no such languages. The few examples, such as OBERON (Mössenböck 1993), MODULA-3 (Harbinson 1992), or ADA 95, result from the evolution of non-object-oriented languages and object orientation is not their main feature. In static typing, commonly used pure object-oriented languages, such as C++ or EIFFEL (Meyer 1992, 1997), offer the programmer plain multiple inheritance. More recent languages like JAVA and C# offer a limited form of multiple inheritance, whereby classes are in single inheritance and types, i.e. *interfaces*, are in *multiple subtyping*. Furthermore, the absence of multiple subtyping was viewed as a deficiency of the ADA 95 revision, and this feature was incorporated in the next version (Taft et al. 2006). This is a strong argument in favour of the importance of multiple inheritance. So there is a real need for efficient object implementation in the context of multiple inheritance and static typing. The multiple inheritance requirement is less urgent in the context of dynamic typing—an explanation is that the canonical static type system corresponding to a language like SMALLTALK (Goldberg and Robson 1983) would be that of JAVA, i.e. multiple subtyping. Anyway, dynamic typing gives rise to implementation issues which are similar to that of multiple inheritance, even though the solutions are not identical, and the combination of both, as in CLOS (Steele 1990), hardly worsens the situation. In this article, we focus on static typing and multiple inheritance. Hence, our target languages can be thought of as C++, JAVA, C# or EIFFEL.

Besides implementation techniques, which are concerned with low-level data structures and code sequences, the overall run-time efficiency strongly depends on what we call, here, *compilation schemes*, that involve the production of an executable from the source code files and include various processors like compilers, linkers and loaders. We consider that the object-oriented philosophy is best expressed under the *open world assumption* (OWA)—each class must be designed and implemented while ignoring how it will be reused, especially whether it will be specialized in single or multiple inheritance. OWA is ensured by *separate compilation* and *dynamic loading*. However, as JAVA and C++ exemplify it, we do not know any implementation of multiple inheritance under the OWA that would be perfectly efficient and scalable, i.e. time-constant and space-linear. In contrast, the *close world assumption* (CWA), that is ensured by *global compilation*, allows for both efficient implementations and various optimizations that partly nullify the overhead of late binding. This approach is exemplified by the GNU EIFFEL compiler (Zendra et al. 1997; Collin et al. 1997). A variety of combinations stands between these two extremes. For instance, the program elements can be separately compiled under the OWA while the executable is produced by an optimized global linker (Boucher 2000; Privat and Ducournau 2005). Alternatively, some parts of the program—the libraries—can be separately compiled under the OWA, whereas the rest is globally compiled under the CWA. A last example is given by *adaptive compilers* (Arnold et al. 2005) that can be thought of as separate compilation under temporary CWAs that can be questioned when further loading invalidates the assumptions—partial recompilation is thus required. In this paper, we do not consider *adaptive compilers* and we mostly consider compilation schemes that do not involve any recompilation.

Implementation techniques and compilation schemes are closely related—when excluding recompilations, not all

pairs are compatible. Moreover, compilation schemes can be ordered from full OWA to full CWA and the compatibility of techniques w.r.t. schemes is monotonic—when a technique is compatible with a scheme, it is also compatible with all schemes that are more closed than the considered one. In principle, language specifications should be independent from implementation. However, in practice, many language specifications are closely dependent on a precise implementation technique or compilation scheme. For instance, the `virtual` keyword makes C++ inseparable from its subobject-based implementation (Ellis and Stroustrup 1990; Lippman 1996), whereas EIFFEL cannot be considered other than with global compilation, because of its unrestricted covariance which would yield unsafe and inefficient code with separate compilation. Therefore, an objective comparison of the respective efficiency of these languages is almost impossible.

From the beginning of object-oriented programming, many implementation techniques have been proposed. Some of them are commonly used in production run-time systems, in JAVA and C# virtual machines or C++ and EIFFEL compilers. Many others have been studied in theory, their time-efficiency may have been assessed in an abstract framework like (Driesen 2001) and their space-efficiency may have been tested on some benchmarks made of large class hierarchies. Most often, however, no empirical assessment has been made or, alternatively, the empirical assessment of the considered technique did not allow a fair comparison with alternative techniques, with *all other things being equal*. There are many reasons to such a situation. Implementing an object-oriented language is hard work and implementing alternative techniques is markedly harder—the compiler needs an open architecture and fair measurements require a perfect reproducibility.

So this article is a step in a project that intends to produce fair assessment of various alternative implementation techniques, with *all other things being equal*. The previous steps included abstract analysis in the Driesen's framework, and simulation of the memory occupation based on large scale benchmarks (Ducournau 2002, 2006, 2008). In the past few years, we developed a new language, called PRM, and a compiler with an open modular architecture which makes it easy to test alternatives techniques. Early results presented empirical measures of program efficiency, but the tested programs were artificial (Privat and Ducournau 2005). In this article, we present an empirical assessment of the time-efficiency of a real program, according to the underlying implementation techniques and compilation schemes that are used to produce the executable, and on a variety of processors. Our testbed consists of the PRM compiler, which compiles PRM source code to C code and is applied to itself. The tested techniques include: (i) *coloring* (Ducournau 2006) which represents the extension of the single-subtyping implementation to multiple inheritance under partial CWA; (ii) *binary tree dispatch* (BTD) (Zendra et al. 1997; Collin et al. 1997) which requires stronger CWA; (iii) C++ *subobjects*, (iv) *perfect hashing* (Ducournau 2008) that has been recently proposed for JAVA interfaces under pure OWA, (v) *incremental coloring* (Palacz and Vitek 2003) also proposed for JAVA interfaces, that is an incremental version of coloring which requires load-time recomputations, (vi) *caching*, when it is coupled with the less efficient techniques.

The contribution of this article is thus reliable time measurements of different executables produced from the same program benchmark, according to different implementations and compilations. From a qualitative standpoint, the conclusions are not new—our tests mostly confirm the intuitive or theoretical abstract assessments of the tested approaches. As expected, efficiency increases as CWA strengthens. However, from a quantitative standpoint, the conclusions are quite new—these tests represent, to our knowledge, the first systematic comparisons between very different approaches with *all other things being equal*. Among others, these tests give the first empirical assessment of: (i) both the oldest and newest techniques—subobjects and perfect hashing; (ii) the overhead of OWA vs. CWA; (iii) the overhead of multiple vs. single inheritance; and (iv) the first empirical comparison between C++ and EIFFEL implementations.

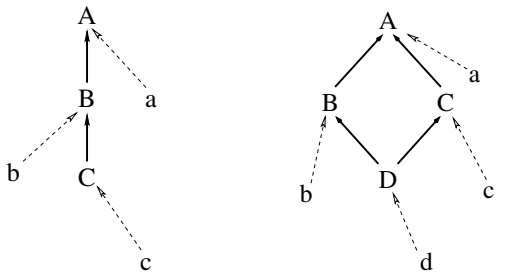
This article is structured as follows. Section 2 surveys the implementation techniques that are tested here and discusses their expected efficiency. Section 3 presents compilation schemes and their compatibility with the different implementation techniques. Section 4 describes the testbed and some statistics on the tested program, then discusses the precise experimental protocol, its reliability and reproducibility. Section 5 presents the time measures and discusses the relative overhead of the different combinations. Finally, the last section presents related works, first conclusions and prospects.

2. Implementation Techniques

Implementation techniques are concerned with object representation, that is, the object layout and the associated data structures that support method invocation, attribute access and subtype testing.

2.1 Single Subtyping

In separate compilation of statically typed languages, late binding is generally implemented with method tables, aka *virtual function tables* (VFT) in C++ jargon. Method calls are then reduced to function calls through a small fixed number (usually 2) of extra indirections. An object is laid out as an attribute table, with a pointer at the method table. With single inheritance, the class hierarchy is a tree and the tables implementing a class are straightforward extensions of those of its single direct superclass (Figure 2). The resulting implementation respects two essential *invariants*: (i) a *reference* to an object does not depend on the static type of the reference;



Two class hierarchies with associated instances, in single (left) and multiple (right) inheritance—solid arrows represent class specialization and dashed arrows represent instantiation.

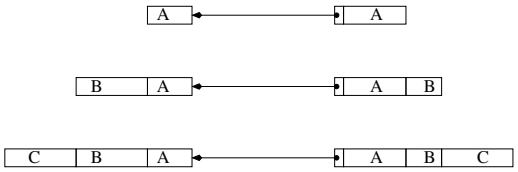
Figure 1. Single and multiple inheritance hierarchies

(ii) the *position* of attributes and methods in the table does not depend on the dynamic type of the object. Therefore, all accesses to objects are straightforward. This simplicity is due to both static typing and single inheritance—dynamic typing adds the same kind of complication as multiple inheritance, since the same property name may be at different places in unrelated classes. So this accounts for method invocation and attribute access under the OWA.

Regarding subtype testing, the technique proposed by (Cohen 1991) also works under the OWA. It involves assigning a unique ID to each class, together with an invariant position in the method table, in such a way that an object x is an instance of the class C if and only if the method table of x contains the class ID of C, at a position uniquely determined by C. Readers are referred to (Ducournau 2008) for implementation details.

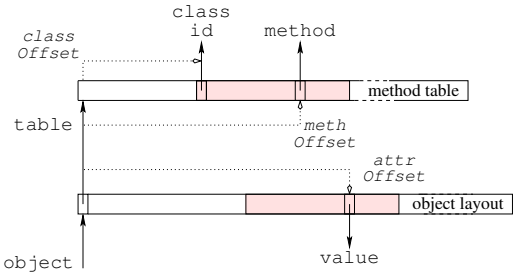
In this implementation, the total table size is roughly linear in the cardinality of the specialization relationship, i.e. linear in the number of pairs (x, y) such that x is a subtype (subclass) of y ($x \preceq y$). Cohen's display uses exactly one entry per such pair and the total table size is linear if one assumes that methods and attributes are uniformly introduced in classes. Moreover, the size occupied by a class is also linear in the number of its superclasses. More generally, linearity in the number of classes is actually not possible since efficient implementation requires some compilation of inheritance, i.e. some superclass data must be copied in the tables for subclasses. Therefore, usual implementations are, in the worst case (i.e. deep rather than broad class hierarchies), quadratic in the number of classes, but linear in the size of the inheritance relationship. The inability to do better than linear-space is likely a consequence of the constant-time requirement. As a counter-example, (Muthukrishnan and Muller 1996) propose an implementation of method invocation with $\mathcal{O}(N + M)$ table size, but $\mathcal{O}(\log \log N)$ invocation time, where N is the number of classes and M is the number of method definitions.

The three mechanisms that we consider—namely method invocation, attribute access and subtype testing—would seem to be equivalent, as they reduce to each other. Obviously, method tables are object layout *at the meta-level*.



The single subtyping implementation of the example from Fig. 1-left. Object layouts (right) are drawn from left to right and method tables (left) from right to left. In the object layouts (resp. method tables) the name of a class represents the set of attributes (resp. methods) introduced by the class.

Figure 2. Single subtyping implementation



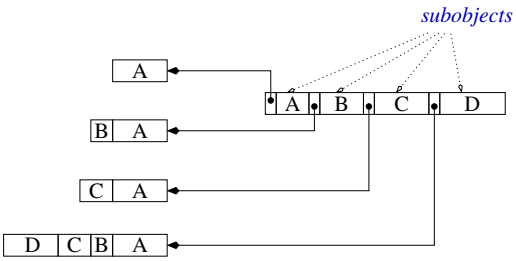
The diagram depicts the precise object representation for all three mechanisms. Pointers and pointed values are in roman type with solid lines, and offsets are italicized with dotted lines. Each mechanism relies on a single invariant offset. The grey parts represent the groups of attributes and methods introduced by a given class. Cohen's display amounts to reserving an entry in the method group for the class ID.

Figure 3. Object representation in single subtyping

So, apart from memory-allocation considerations, they are equivalent. Moreover, an attribute can be read and written through dedicated *accessor* methods—hence, attribute access can always reduce to method invocation (Section 2.7). An interesting analogy between subtype tests and method calls can also be drawn from Cohen's display. Suppose that each class C introduces a method `amIaC?` that returns `yes`. In dynamic typing, calling `amIaC?` on an unknown receiver x is exactly equivalent to testing if x is an instance of C—in the opposite case, an exception will be signaled. In static typing, the analogy is less direct, since a call to `amIaC?` is only legal on a receiver statically typed by C, or a subtype of C—this is type safe but quite tautological. However, subtype testing is inherently type unsafe and one must understand `amIaC?` as a pseudo-method, which is actually not invoked but whose presence is checked. The test fails when this pseudo-method is not found, i.e. when something else is found at its expected position. This informal analogy is important—it implies that one can derive a subtype testing implementation from *almost* any method call implementation. We actually know a single counter-example, when the implementation depends on the static type of the receiver, as in subobject-based implementations (Section 2.2).

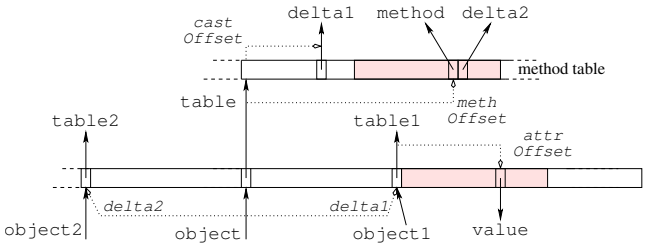
2.2 Subobjects (SO)

With multiple inheritance, both invariants of reference and position cannot hold together, at least if compilation—i.e. computation of positions—is to be kept separate. For in-



Object layout and method table of a single instance of the class D from the diamond example of Fig. 1-right.

Figure 4. Subobject-based implementation



The diagram depicts the precise object representation restricted to method invocation, attribute access and pointer adjustemnt. `object` is the current reference to the considered object. `delta1` is the pointer adjustment that is required for going from `object` to `object1` subobjects, e.g. for accessing an attribute defined in the class corresponding to the latter. `delta2` is the pointer adjustment that is required for going from `object` subobject to that of the class which defines the invoked method.

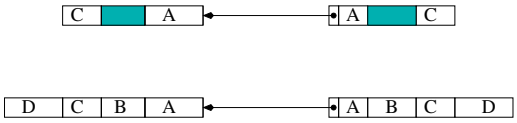
Figure 5. Object representation with subobjects

stance, in the diamond hierarchy of Figure 1-right, if the implementations of B and C simply extend that of A, as in single inheritance, the same offsets will be occupied by different properties in B and C, thus prohibiting a sound implementation of D. Therefore, the 'standard' implementation of multiple inheritance in a static typing and separate compilation setting—i.e. that of C++—is based on *subobjects* (SO). The object layout is composed of several subobjects, one for each superclass of the object's class. Each subobject contains attributes *introduced* by the corresponding class, together with a pointer to a method table which contains the methods *known* by the class (Fig. 4 and 5). Both invariants are dropped, as both reference and position depend on the current static type. This is the C++ implementation, when the keyword `virtual` annotates each superclass (Ellis and Stroustrup 1990; Lippman 1996; Ducournau 2002). It is time-constant and compatible with dynamic loading, but method tables are no longer space-linear. The number of method tables is exactly the size of the specialization relationship. When a class is in single inheritance, its total table size is itself quadratic in the number of superclasses—so, in the worst case, the total size for all classes is cubic in the number of classes. Furthermore, all polymorphic object manipulations—i.e. assignments and parameter passing, when the source type is a strict subtype of the target type—which are quite numerous, require *pointer adjustments* between source and target types, as they correspond to different subobjects. These pointer adjustments are purely mechanical and do not bring any semantics. They are also safe—i.e. the target type is always a supertype of the source type—so they are implemented more efficiently than subtyping tests. They can be done with explicit pointers, called VBPTRs, in the object layout or with offsets in the method tables. There are, however, a lot of variants, according to whether compiler-generated fields are allocated in the object layout, like VBPTRs, or in the method tables. (Sweeney and Burke 2003) analyse this variety, from the ARM implementation, where all fields are allocated in the object layout, to the ALL implementation, where all fields are allocated in the method tables. Although VBPTRs are markedly more time-efficient since they save an access to method table at each pointer adjustment, they are also over space-consuming. Therefore, we only consider ALL implementation here, which is closest to most actual implementations. Furthermore, contrary to single inheritance, there is no known way of deriving a subtype test from the technique used for method invocation. It is no longer possible to consider that testing if an object is an instance of some class C is a kind of method introduced by C because this pseudo-method would not have any known position other than in static subtypes of C. So, in our tests, we will complete subobject-based implementation with perfect hashing (Section 2.5) for subtype testing.

Empty-subobject optimization (ESO) (Ducournau 2002) represents further improvement that applies when a class does not introduce any attribute—hence the corresponding subobject is empty—especially when it does not either introduce any method and have a single direct superclass. In this case, both subobjects can be merged and the statistics presented in the aforementioned article show that the improvement is significant. Although the designers of C++ compilers do not seem to be aware of the possibility of ESO, we have used it in our tests because it was required in the PRM testbed for efficient *boxing* and *unboxing* of primitive types—unlike C++ and like JAVA 1.5 and EIFFEL, the PRM type system consider that primitive types are subtypes of some general types like `Object`.

Subobjects can also apply to JAVA interfaces, with an improved empty-subobject optimization that relies on class single inheritance. The technique, detailed in (Ducournau 2002) after the bidirectional layout of (Myers 1995), is space-linear contrary to general subobject-based implementation. It would be interesting to test it but it is incompatible with both our PRM testbed, because of the distinction between classes and interfaces, and current JVMs, because it is not reference-invariant.

When the `virtual` keyword is not used—we call it *non-virtual* inheritance—the C++ implementation is markedly more efficient but no longer fully reusable because it yields *repeated inheritance*—so the language is no longer compatible with both multiple inheritance and dynamic loading. As coloring is certainly more efficient than C++ non-virtual im-



With the same class diamond as in Fig. 1 and 4, implementation of A and B is presumed to be the same as in Fig. 2. So the implementation of C leaves some empty place (in grey) for B in anticipation of D. The final object representation is the same as in Fig. 3.

Figure 6. Coloring implementation

plementation, we do not test the latter and, in the following, we only consider C++ under the *virtual* implementation.

2.3 Coloring (MC/AC)

The coloring approach is quite versatile and naturally extends the single inheritance implementation to multiple inheritance, while meeting all requirements except compatibility with dynamic loading. The technique takes its name from *graph coloring*, as its computation amounts to coloring some particular graph[2]. *Method coloring* was first proposed by (Dixon et al. 1989) under the name of *selector coloring* for method invocation. (Pugh and Weddell 1990) and (Ducournau 1991) applied *coloring* to attribute access and (Vitek et al. 1997) to subtype testing (under the name of *pack encoding*). Hereafter, MC denotes coloring when used for method invocation and subtype testing, and AC denotes attribute coloring.

The general idea of coloring is to keep the two *invariants* of single inheritance, i.e. *reference* and *position*. An injective numbering of attributes, methods and classes verifies the position invariant, so this is clearly a matter of optimization for minimizing the size of all tables—or, equivalently, the number of *holes*, i.e. empty entries. However, this optimization cannot be done separately for each class, it requires a global computation for the whole hierarchy. The problem of minimizing the total table size is akin to the *minimum graph coloring* problem (Garey and Johnson 1979). Like minimal graph coloring, the coloring problem considered here has been proven to be NP-hard in the general case. Therefore heuristics are needed and various experiments have shown their efficiency and that the technique is tractable. Finally, an important improvement is *bidirectionality*, introduced by (Pugh and Weddell 1990), which involves using positive and negative offsets and reduces the hole number. Figure 6 depicts the implementation yielded by unidirectional coloring in the diamond example from Figure 4. The implementation of classes A and B is presumed to be identical to that of Figure 2. Hence, computing the tables for C must reserve some space for B in the tables of D, their common subclass. Thus, some holes appear in the C tables and these holes are filled, in D, by all data specific to B. In bidirectional coloring, all holes would have been saved by placing C at negative offsets.

A detailed presentation of coloring is beyond the scope of this paper and readers are referred to (Ducournau 2006) which reviews the approach. The point to get is 2-fold: (i) in practice, object layout, method tables and code sequences are exactly those of single subtyping, except for the presence of holes; (ii) this is obtained by rather sophisticated algorithms which require complete knowledge of the class hierarchy. Actually, we have exchanged multiple inheritance for dynamic loading.

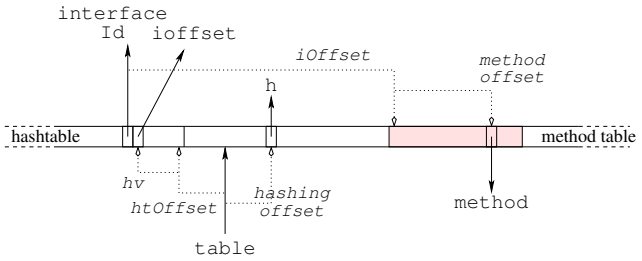
2.4 Binary Tree Dispatch (BTD)

Not all object-oriented implementations are based on method tables. In SMART EIFFEL, the GNU EIFFEL compiler, method tables are not used. Instead, objects are tagged by their class identifier and all three mechanisms—particularly method invocation—are implemented using balanced binary dispatch trees (Zendra et al. 1997; Collin et al. 1997). However, the approach is practical only because compilation is global, hence all classes are statically known. Furthermore, type analysis restricts the set of *concrete types* (Bacon and Sweeney 1996; Grove and Chambers 2001) and makes dispatch efficient. BTD is also an interesting example of the possible disconnection between code length, hence inlining, and time efficiency. Indeed, here, both values are in an exponential relationship—hence proving that not all efficient code sequences are inlinable. Anyway, BTD is not *time-constant*.

The efficiency of BTDs relies on the conditional branching prediction of modern processors. Thanks to their pipeline architecture, the cost of well-predicted branchings is free. On the contrary, mispredictions break the pipe and cost about 10 cycles or more, and most undirect branches are mispredicted—so this misprediction cost holds for all VFT-based techniques. Readers are referred to (Driesen 2001) for a more in-depth analysis. An overall consequence is that BTDs are statistically more efficient than VFTs when the number of tests is small. It however depends on the statistical distribution of dynamic types on each call site—it is easy to construct worst-case artificial programs whereby all predictions fail, making VFTs far better than BTDs. In the following, BTD_i will denote BTDs of depth bounded by i. BTD_0 corresponds to static calls and BTD_∞ denotes unbounded BTDs.

Overall, BTDs are efficient when the number of competing methods is low—the corresponding call sites are often called *oligomorphic*—but coloring should be preferred when this number is higher—*megamorphic* call sites. An interesting tradeoff involves combining BTD_k and coloring, with $k = 3$ or 4. This makes the resulting technique constant-time and inlinable. Furthermore, method tables are restricted to the methods that have a megamorphic call site, hence at least 2^k implementations. BTDs also apply to subtype test-

[2] This graph is a *conflict graph* with a vertex for each class and an edge between any two vertices that have a common subclass and thus must have their attributes (resp. methods or class IDs) stored at distinct offsets, since attributes (resp. methods or class IDs) of both classes coexist in objects (resp. method tables) of the common subclass.



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

Figure 7. Perfect hashing for JAVA interfaces

ing and attribute access but, in the context of global compilation, coloring is likely better.

2.5 Perfect Hashing (PH)

In a recent article (Ducournau 2008) we proposed a new technique based on perfect hashing for subtype testing in a dynamic loading setting. The problem can be formalized as follows. Let $(X, \preceq)$ be a partial order that represents a class hierarchy, namely X is a set of classes and $\preceq$ the specialization relationship that supports inheritance. The subtype test amounts to checking at run-time that a class c is a superclass of a class d, i.e. $d \preceq c$. Usually d is the dynamic type of some object and the programmer or compiler wants to check that this object is actually an instance of c. The point is to efficiently implement this test by precomputing some data structure that allows for constant time. Dynamic loading adds a constraint, namely that the technique should be inherently incremental. Classes are loaded at run-time in some total order that must be a *linear extension* (aka *topological sorting*) of $(X, \preceq)$—that is, when $d \prec c$, c must be loaded before d.

The *perfect hashing* principle is as follows. When a class c is loaded, a unique identifier id_c is associated with it and the set $I_c = \{id_d \mid c \preceq d\}$ of the identifiers of all its superclasses is known—if needed, yet unloaded superclasses are recursively loaded. So, $c \preceq d$ iff $id_d \in I_c$. This set I_c is immutable, hence it can be hashed with some *perfect hashing function* h_c, that is, a hashing function that is injective on I_c (Sprugnoli 1977; Czech et al. 1997). The previous condition becomes: $c \preceq d$ iff $ht_c[h_c(id_d)] = id_d$, whereby ht_c denotes the hashtable of c. Moreover, the cardinality of I_c is denoted n_c. The technique is incremental since all hashtables are immutable and the computation of ht_c depends only on I_c. The perfect hashing functions h_c are such that $h_c(x) = hash(x, H_c)$, whereby H_c is the hashtable size defined as the least integer such that h_c is injective on I_c.

Two functions were considered for $hash$, namely *modulus* (noted `mod`) and bit-wise `and`[3]. The corresponding techniques are denoted hereafter PH-`mod` and PH-`and`. However, these two functions involve a tradeoff between space and time efficiency. The former yields more compact tables but the integer division latency may be more than 20 cycles, whereas the latter is a 1-cycle operation but yields larger tables. In a forthcoming paper (Ducournau and Morandat 2009), we improve the technique with a new hashing function that combines bit-wise `and` with a `shift` for truncating trailing zeros (PH-`and+shift`)—it reduces the total hashtable size at the expense of a few extra instructions that are expected to be run in parallel.

To our knowledge, PH is the only constant-time technique for subtype testing that allows for both multiple inheritance and dynamic loading at reasonable spatial cost. In a static typing setting, the technique can also be applied to method invocation and we did propose, in the aforementioned article, an application to JAVA interfaces. For this, the hashtable associates, with each implemented interface, the offset of the group of methods that are introduced by the interface. Figure 7 recalls the precise implementation in this context. The method table is bidirectional. Positive offsets involve the method table itself, organized as with single inheritance. Negative offsets consist of the hashtable, which contains, for each implemented interface, the offset of the group of methods introduced by the interface. The object header points at its method table by the `table` pointer. `#hashingOffset` is the position of the hash parameter (`h`) and `#htOffset` is the beginning of the hashtable. At a position `hv` in the hashtable, a two-fold entry is depicted that contains both the implemented interface ID, that must be compared to the target interface ID, and the offset `iOffset` of the group of methods introduced by the interface that introduces the considered method. The table contains, at the position `#methodOffset` determined by the considered method in the method group, the address of the function that must be invoked. To our knowledge, PH is, together with C++ subobject-based implementation, the only constant-time technique for method invocation that allows for both multiple inheritance and dynamic loading at reasonable spatial cost.

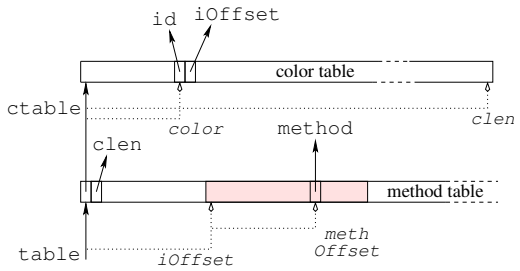
2.6 Incremental Coloring (IC)

An incremental version of coloring (denoted IC) has been proposed by (Palacz and Vitek 2003) for implementing interface subtype testing in JAVA. An application to method invocation in the same style as for PH has been proposed in (Ducournau 2008). As coloring does not work under the OWA, IC can require some load-time recomputations. So its data structures involve extra indirections and several unrelated memory locations that should increase cache misses (Figure 8). Readers are referred to (Ducournau 2008) for detailed implementation and discussion.

2.7 Accessor Simulation (AS)

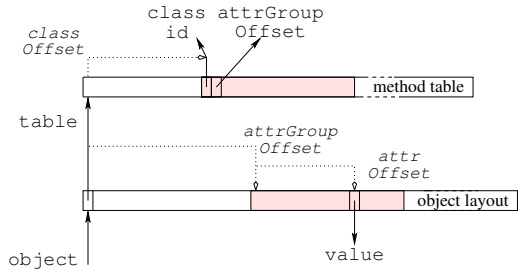
An accessor is a method that either reads or writes an attribute. Suppose that all accesses to an attribute are through an accessor. Then the attribute layout of a class does not have to be the same as the attribute layout of its superclass. A class

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



The implementation resembles PH, apart from the fact that the interface position is invariant instead of being the result of specific hashing. Moreover, the recomputable color table requires an extra indirection, together with its size (`clen`) and bound checking, and the color itself requires memory access (not represented in the diagram).

Figure 8. Incremental coloring for JAVA interfaces



The diagram depicts the precise object representation with accessor simulation coupled with class and method coloring, to be compared with Fig. 3. The offset of the group of attributes introduced by a class (`attrGroupOffset`) is associated with its class ID in the method table and the position of an attribute is now determined by an invariant offset (`attrOffset`) w.r.t. this attribute group.

Figure 9. Accessor simulation with method coloring

will redefine the accessors for an attribute if the attribute has a different offset in the class than it does in the superclass. True accessors require a method call for each access, which can be inefficient. However, a class can simulate accessors by replacing the method address in the method table with the attribute's offset. This approach is called *field dispatching* by (Zibin and Gil 2003). Another improvement is to group attributes together in the method table when they are introduced by the same class. Then one can substitute, for their different offsets, the single relative position of the attribute group, stored in the method table at an invariant position, i.e. at the class color with coloring—Fig. 9 (Myers 1995; Ducournau 2002). In PH and IC, the attribute-group offset is associated with the class ID and method-group offset in the hash- or color-table, yielding 3-fold table entries.

Accessor simulation is a generic approach to attribute access which works with any method invocation technique—only grouping can be conditioned by static typing, since attributes must be partitioned by the classes which introduce them. It is, however, meaningless to use it with subobject-based implementation (SO) which provides two different accesses to attributes according to whether the receiver's static type (`rst`) is the attribute introduction class (`aic`) or not. The former is identical to attribute coloring (AC), whereas the latter is identical to accessor simulation (AS) with method coloring (MC). For instance, in Fig. 5, `rst`$\neq$`aic`.

Among the various techniques that we have described, some apply only to method invocation and subtype testing, e.g. perfect hashing and incremental coloring. Hence, these techniques can serve for JAVA interface implementation. Accessor simulation is a way of applying them to full multiple inheritance. It can also replace attribute coloring, if holes in object layout are considered to be over space-consuming.

2.8 Caching and Searching (CA)

A common implementation policy that is often used with dynamic typing or JAVA interfaces involves some naive and rather inefficient implementation technique coupled with caching for memoizing the results of the last search. For instance, with JAVA interfaces, each method table will cache the class ID and the interface offset of the last succeeding access (Alpern et al. 2001a,b; Click and Rose 2002). Of course this cache might serve for any table-based subtyping technique and for all three mechanisms, at the expense of caching three data, namely class ID and method and attribute group offsets. Obviously, the improvement is a matter of statistics and those presented in (Palacz and Vitek 2003) show that, according to the different benchmarks, cache miss rates can be as low as 0.1% or more than 50%. In our tests, we will also consider PH and IC when they are coupled with caching—one might expect, for instance, that caching degrades PH-`and` but improves PH-`mod`. Like IC and unlike all other techniques, caching requires method tables to be writable, hence allocated in data memory segments.

3. Compilation Schemes

Compilation schemes represent the production line of executable programs from the source code files. They can involve various processors such as compilers, linkers, virtual machines, loaders, just-in-time compilers, etc.

3.1 Under Pure OWA—Dynamic Loading (D)

As aforementioned, object-oriented philosophy, especially reusability, is best expressed by the OWA. Pure OWA corresponds to *separate compilation* and *dynamic loading*—this scheme will be denoted D hereafter. Under the OWA, a class C—more generally, a code unit including several classes—is compiled irrespective of the way it will be used in different programs, hence ignoring its possible subclasses and *clients*[4]. On the contrary, a subclass or a client of C must know the "model" (aka "schema") of C, which contains the interface of C possibly augmented by some extra data—e.g. it is not restricted to the *public* interface. This class model is included in specific header files (in C++) or automatically extracted from source or compiled files (in JAVA). Without loss of generality, it can be considered as an instance of some

[4] A client of C is a class that uses C or a subclass of C, as a type annotation (e.g. $x : C$) or for creating instances (`new` C).

metamodel (Ducournau and Privat 2008). The code itself is not needed.

Separate compilation is a good answer to the modularity requirements of software engineering—it provides speed of compilation and recompilation together with locality of errors, and protects source code from both infringement and hazardous modifications. With separate compilation, the code generated for a program unit, here a class, is correct for all correct future uses.

3.2 Under Pure CWA—Global Compilation (G)

Complete knowledge of the whole class hierarchy offers many ways to efficiently implement multiple inheritance. CWA presents several gradual advantages: (i) the class hierarchy is closed and the models of all classes can be known as a whole; (ii) the code of each class is also known; (iii) the program entry point can also be known.

In all cases, some *static type analysis* is possible, from a simple *class hierarchy analysis*[5] (Dean et al. 1995b), when only (i) holds, to more sophisticated algorithms (Bacon and Sweeney 1996; Grove and Chambers 2001) when all three points hold. With global compilation (scheme denoted G), when the program entry point is known and the language does not provide any metaprogramming facility, type analysis can precisely compute the receiver's *concrete type* at each call site, making it easy to identify mono-, oligo- and megamorphic sites, so that each category can be implemented with the best technique, i.e. static calls, BTD_i and coloring. Other well-known algorithms are RTA (Bacon et al. 1996) and CFA (Shivers 1991). Moreover, dead code can be ruled out and other optimizations like *code specialization* (Dean et al. 1995a; Tip and Sweeney 2000) can further reduce polymorphism—the former decreases the overall code size but the latter increases it. We do not consider them here.

3.3 Separate Compilation, Global Linking (S)

The main defect of coloring is that it requires complete knowledge of all classes in the hierarchy. This complete knowledge could be achieved by global compilation. However, leaving the modularity provided by separate compilation may be considered too high a price for program optimization. An alternative was already noted by (Pugh and Weddell 1990). Coloring does not require knowledge of the code itself (point (ii)), but only of the model of the classes (point (i)), all of which is already needed by separate compilation. Therefore, the compiler can separately generate the compiled code without knowing the value of the colors of the considered entities, representing them with specific symbols. At link time, the linker will collect the models of all classes and color all the entities, before substituting values to the different symbols, as a linker commonly does. The linker must also generate method tables.

[5] '*Class hierarchy analysis*' is a common term that denotes any analysis of the class hierarchy. It is however also the label (CHA) of the specific analysis proposed by (Dean et al. 1995b).

	Scheme				
	D	S	O	H	G
SO	•	•	∗	∗	∗
PH	•	•	∗	∗	∗
IC	•	•	∗	∗	∗
CA	•	•	∗	∗	∗
MC	×	•	•	∗	•
BTD	×	×	•	∗	•
AC	×	•	•	∗	•
AS	•	•	•	∗	•

•: Compatible and tested, ∗: Compatible but non-tested, ×: Incompatible

Table 1. Compatibility between compilation schemes and implementation techniques

3.4 Separate Compilation, Global Optimization (O)

(Privat and Ducournau 2005) propose a mixed scheme which relies on some link-time type analysis. As the class hierarchy is closed, CHA can be applied, that will determine whether a call site is monomorphic or polymorphic. Link-time optimization is possible if, at compile-time, the code generated for a call site is replaced by a call to a special symbol, which is for instance formed by the name of the considered method and the static type of the receiver. Then, at link-time, a stub function—called a *thunk* like in C++ implementations (Lippman 1996)—is generated when the call site is polymorphic. For monomorphic sites, the symbol is just replaced by the name of the called procedure, thus yielding a static call.

More sophisticated type analyses are possible if a model of internal type flow, called an *internal model*—in contrast, the model discussed in Section 3.1 is called *external model*—is generated at compile time (Privat and Ducournau 2005). (Boucher 2000) proposed a similar architecture in a functional programming setting.

Another hybrid scheme (H) would involve separate compilation of common libraries, coupled with global compilation of the specific program and global optimization of the whole.

3.5 Compilation vs. Implementation

Table 1 presents the compatibility between implementation techniques and compilation schemes. Table 2 recalls the expected efficiency that can be deduced from previous abstract studies. Efficiency must be assessed from the space and time standpoints. Space-efficiency assessment must consider code length, static data (i.e. method tables) and dynamic data (i.e. object layout). Time-efficiency assessment must consider run- and compile-time together with load- or link-time.

Not all compatible combinations have been tested because many of them are not interesting. For instance, all techniques that are compatible with the OWA are less efficient than coloring and BTD. So testing them in O, H and G schemes would be wasting time. Moreover, for these techniques, there is no difference between D and S. Hence, S is the right scheme for comparing the efficiency of implemen-

	Space			Time		
	Code	Static	Dyn.	Run	Compile	Load/Link
SO	–	– –	– –	–	++	++
IC	–	+	+++	–	++	– –
PH-and	–	–	+++	–	++	+
PH-mod	–	+	+++	– –	++	+
PH-and +CA	– –	– –	+++	– –	++	+
PH-mod +CA	– –	–	+++	–	++	+
MC	++	++	+++	++	+	–
$BTD_{i<2}$	+++	+++	+++	+++	++	– –
$BTD_{i>4}$	– – –	+++	+++	– – –	–	– –
AC	++	++	+	++	+	–
AS	+	+	+++	–	+	+

+++: optimal, ++: very good, +: good, –: bad, – –: very bad, – – –: unreasonable

Table 2. Expected efficiency

tation techniques like SO, PH, IC and MC. O and G are the right ones for comparing MC and BTD. Moreover, the comparison can also consider various type analysis algorithms (CHA, RTA or CFA) and polymorphism degrees for BTDs.

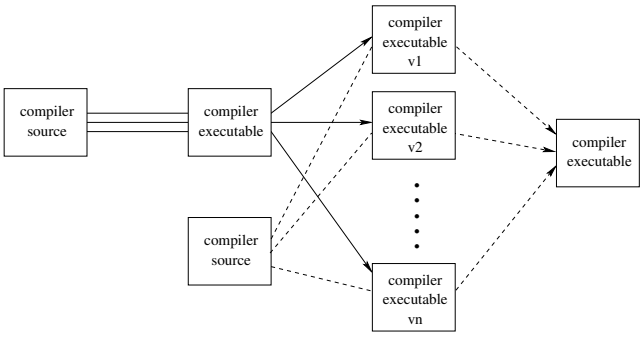
AC and AS can be compared in all schemes but D and the comparison closely depends on the underlying method invocation technique. Coupling AC with PH or IC is, however, possible and provides an assessment of the use of the considered method invocation technique in the restricted case of JAVA interfaces. On the contrary, coupling these techniques with AS amounts to considering them in a full multiple inheritance setting. In contrast, the H scheme has not been tested, partly for want of time, and partly because of the difficulty of distinguishing between libraries and program.

4. Compilation Testbed

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

Tested Program. We have implemented all these techniques in the PRM compiler, which is dedicated to exhaustive assessment of various implementation techniques and compilation schemes (Privat and Ducournau 2005; Morandat et al. 2009). The benchmark program is the compiler itself, which is written in PRM and compiles the PRM source code into C code. There are a lot of compilers in the picture, so Figure 10 depicts the precise testbed. In these tests, the C code generated by the PRM compiler and linker is the code of the considered techniques in the considered compilation scheme. So the code can be generated at compile or link time according to the scheme.

The PRM compiler is actually not compatible with dynamic loading (D) but the code for PH or IC has been generated in separate compilation (S) exactly as if it were generated at load-time, with hash parameters and tables being computed at link-time. In that case, all link-time optimizations are deactivated. Hence, although these tests represent a kind of simulation, they must be quite reliable. Only the effect of cache misses is likely underestimated, especially for



Some compiler source is compiled by some compiler executable, according to different options, thus producing different variants v_1, .., v_n, of the same executable compiler (solid lines). Another compiler source (possibly the same) is then compiled by each variant, with all producing exactly the same executable (dashed lines), and the duration of this compilation (i.e. the dashed lines) is measured.

Figure 10. The PRM testbed

incremental coloring—all color tables are here allocated in the same memory area, whereas load-time recomputations should scatter them in the heap.

Table 3 presents the static characteristics of the tested program, i.e. the PRM compiler, namely the number of different entities that are counted at compile-time, together with the run-time invocation count for each mechanism. Of course, these numbers do not depend on compilation variants, though some of them concern only some specific variants. The Table details statistics of method call sites according to their polymorphism degree, that is, the BTD_i that can implement them, according to the CHA type analysis. A call site is counted in BTD_i if the cardinality of the receiver's concrete type is between $(2^{i-1}+1)$ and 2^i. Finally, the cache-hit rate has been measured when coupling perfect hashing (PH) with caching (CA)—of course, it does not depend on the precise technique that is associated with CA. The measured cache-hit rate is about 50% with attribute coloring (AC) but only 34% with accessor simulation (AS). The latter is markedly lower than those reported by (Palacz and Vitek 2003), but this is simply explained by the fact that the cache is used here for all classes and several mechanisms, whereas its use was restricted, in the aforementioned paper, to interfaces and subtype testing. Cache hits and monomorphic calls represent similar but dual data. Monomorphism is a permanent characteristics of a call site that is always used for calling the same method, whereas a cache hit is a momentary characteristics of a method table that is used for an access to the same superclass as in the previous access.

These statistics show that the program size is significant and that it makes intensive usage of object-oriented features. Moreover, the high number of monomorphic calls—about 79 or 64% of calls sites, according to whether static or dynamic statistics are considered—is consistent with statistics that are commonly reported in the literature. Of course, the validity of such experiments that rely on a single program might be questioned. This is however a large program, which is fully object-oriented and intensively uses the basic mecha-

number of		static	dynamic
class	introductions	531	—
	instantiations	6317	36 M
	subtype tests	189	130 M
method	introductions	2577	—
	definitions	4379	—
	calls	14808	2510 M
BTD	0	11719	1607 M
	1	820	62 M
	2	576	424 M
	3	681	23 M
	4	715	28 M
	> 4	297	364 M
attribute	introductions	630	—
	accesses	4323	4285 M
	`rst=aic`	3019	3160 M
	accessor	680	200 M
pointer	adjustments	11471	1668 M
cache hits	with AC	—	52%
	with AS	—	34%

The "static" column depicts the number of program elements (classes, methods and attributes) and the number of sites for each mechanism. The "dynamic" column presents the number of mechanism invocations at run-time (in millions). Method call sites are separately counted according to their polymorphism degree, i.e. the BTD depth that can implement them. Attribute accesses are separately counted when the access is made through an accessor or on a receiver whose static type (`rst`) is the attribute introduction class (`aic`) (Section 2.7). The former is a special case of the latter. Like `rst=aic`, pointer adjustments only concern subobjects (SO)—they consist of all polymorphic assignments and parameter passings, when the source type is a strict subtype of the target type, together with equality tests, when the two types are different. Finally, the cache-hit rate is presented, with or without AS.

Table 3. Characteristics of the tested program

nisms that are tested. Moreover, as the experiments compare two implementations with all other things being equal, the sign of the differences must hold for all programs and only the order of magnitude should vary[6]. This limitation is also inherent to our experimentation. The PRM compiler is the only one that allows such versatility in the basic implementation of object-oriented programs. The counterpart is that the language has been developed with this single goal, so its compiler is the only large-scale program written in PRM.

A last objection can be raised, namely that the PRM compiler might be too inefficient for allowing any firm conclusion. In the following, we consider relative overheads, of the form $(test - ref)/ref$. Of course, if the PRM compiler is one order of magnitude slower than it could be, the results are meaningless. Therefore, we have also to prove that the PRM compiler is not too inefficient. This is, however, much more difficult to prove, since it requires an external comparison that cannot be done with *all other things being equal*. As a reference, we chose the GNU EIFFEL compiler, SMART EIFFEL, that uses global compilation (G) and is considered as very efficient—see Section 2.4. Both languages are fully-fledged object-oriented languages that provide similar features, and both compilers present close characteristics such as type analysis and the same target language. So we compared the compilation time of both compilers, from the source language (EIFFEL or PRM) to C. The compilation times were quite similar, about 60s on the considered processor. Although it does not mean that the PRM compiler is as efficient as SMART EIFFEL, it is however a strong indication that the PRM compiler is not too inefficient. Of course, further improvements will strengthen our results.

Runtime Reproducibility. Tested variants differ only by the underlying implementation technique, with all other things being equal. Moreover, this is true when considering executable files, not only the program logic. Indeed, the compilation testbed is deterministic—that is, two compilations of the same program by the same compiler executable produce exactly the same executable. This means that: (i) the compiler always proceeds along the program text and the underlying object model in the same order; (ii) the memory locations of program fragments, method tables and objects in the heap are roughly the same. So two compiler variants differ only by the code sequences of the considered techniques, all program components occurring in the executables in the same order. Moreover, when applied to some program, two compiler variants v_i and v_j produce exactly the same code—hence, the fact that all dashed arrows point at the same executable (Fig. 10) is not only a metaphor. All program equalities have been checked with the `diff` command on both C and binary files. Overall, the effect of memory locality should be roughly constant, apart from the specific effects due to the considered techniques[7].

However, in spite of the compilation determinism, a compiled program is not exactly deterministic for our fine-grained analysis. Indeed, hash structures are inherently not deterministic when the hashed keys are object addresses. Hence, two runs of the same program can produce different collisions. As hash structures are PRM objects, the precise run-time statistics (column "dynamic" in Table 3) are not exactly reproducible. The variations are actually very low—less than one to a thousand—and do not affect the measures. Furthermore, it does not modify the programm logic because all hash structures used by the PRM compiler are order-invariant—all iterations follow the input order. Overall, considering that the difference between two runs of the same executable is pure noise, we took, for each measure, the minimum value among several tens of runs.

Processors. The tests were performed on a variety of processors (Table 4 and 5):

- I-2, I-4, I-5, I-8 and I-9, from the Intel® Pentium™ family;

[6] This is only true when the comparison focuses on a single parameter, i.e. in the same row or column in Tables 4 and 5.

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

processor	S-1	UltraSPARC III		I-2	Xeon Prestonia		M-3	PowerPC G5		I-4	Xeon Irwindale		I-5	Core T2400	
frequency		1.2 GHz			1.8 GHz			1.8 GHz			2.8 GHz			2.8 GHz	
L2 cache	168.8s	8192 K		111.9s	512 K		89.5s	512 K		53.2s	2048 K		43.6s	2048 K	
year		2001			2001			2003			2006			2006	
technique scheme	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
MC-BTD$_\infty$ CHA G	-27.5	-14.0	18.6	-16.8	-7.7	11.0				-18.9	-3.9	18.4	-14.5	-1.3	15.5
MC-BTD$_2$ CHA G	-28.5	-13.5	21.0	-10.6	-8.1	2.9	-27.9	-12.7	21.0	-17.2	-5.2	14.5	-14.7	-4.3	12.2
MC-BTD$_\infty$ CHA O	***	***	***	-2.8	-2.7	0.2	***	***	***	-2.6	3.2	5.9	4.9	21.4	15.8
MC-BTD$_2$ CHA O	***	***	***	-0.7	-5.0	-4.4	***	***	***	-5.2	0.9	6.4	1.4	15.8	14.2
MC S	0	13.3	13.3	0	1.3	1.3	0	1.4	1.4	0	8.5	8.5	0	13.3	13.3
SO D			–			–			–			–			–
IC D	12.1	40.9	25.7	2.1	8.9	6.7	-2.3	26.6	29.5	4.5	23.3	18.0	6.6	32.9	24.6
PH-and D	16.3	58.3	36.1	3.9	16.1	11.8	7.9	40.0	29.7	4.3	30.0	24.6	8.3	47.6	36.3
PH-and+shift D	17.3	62.5	38.6	6.1	30.6	23.2	16.2	57.4	35.5	8.7	43.2	31.8	16.4	69.6	45.8
PH-mod D	95.6	289.3	99.0	39.7	136.8	69.5	47.2	192.9	99.0	69.2	242.7	102.6	29.0	149.0	93.0
PH-mod +CA D	64.6	245.3	109.7	34.6	160.5	93.6	32.9	157.7	94.0	61.6	262.7	124.4	33.8	155.2	90.7

Each subtable presents the results for a precise processor, with the processor characteristics and the reference compilation time. All other numbers are percentage. Each row describes a method invocation and subtype testing technique. For SO the distinction between AC and AS does not apply. For all other techniques, the first two columns represent the overhead vs pure coloring (MC-AC-S), respectively with attribute coloring (AC) and accessor simulation (AS). The third column is the overhead of accessor simulation vs attribute coloring.
*** Results are currently unavailable

Table 4. Compilation time according to implementation techniques and processors

- A-6 and A-7 are AMD® processors; all x86 are under Linux Ubuntu 8.4 with `gcc` 4.2.4;
- S-1 is a SUN® Sparc™, under SunOS 5.10, with `gcc` 4.2.2;
- M-3 is a Motorola® PowerPC™ G5, under Mac OS X 10.5.3, with `gcc` 4.0.1.

Technical problems with linkers made global optimizations (O) currently unavailable on processors S-1 and M-3. All tests use Boehm's garbage collection (Boehm 1993). The measure itself is done with Unix function `times(2)` which considers only the time of a single process, irrespective of the system scheduler—this is required by multicore technology. Two runs of the same compiler on the same computer should take the same time were it not for the noise produced by the operating system. A solution involves running the tests under single-user boot, e.g. Linux recovery-mode. It has been done for some processors (e.g. I-2, I-4, I-8) but this was actually not possible for remote computers. Finally, a last impediment concerned laptops. Modern laptop processors (e.g. I-5 and I-8) are frequency-variable. The frequency is low when the processor is idle or hot. When running a test, the processor must first warm up before reaching its peak speed, then it finishes by slowing down and cooling. So the peak speed can be very high but only on a short duration. Inserting a pause between each two runs seemed to fix the point and I-8 provides now one of the most steady testbed.

5. Results and Discussion

Tables 4 and 5 presents, for all tested variants and processors, the time measurement and overhead with respect to the full coloring implementation (MC/AC). The last column of Table 5 presents similar statistics of executable size instead of runtime time. Overall, notwithstanding some exceptions that will be discussed hereafter, these tests exhibit many regularities.

Compilation Schemes. Regarding compilation schemes:

- As expected, global compilation (G) is markedly better than separate compilation (S). The high ratio of monomorphic calls mostly explains this result, since the difference between MC-S and MC-BTD$_0$-G results only from monomorphic calls and BTD$_2$ hardly improves it.
- In contrast, link-time optimization (O) does not provide the expected improvement. It would mean that the gain resulting from 64% of static calls is offset by the thunk overhead in the 36% of polymorphic calls. This is unexpected because one might have thought that pipelines would have made the thunk almost free, apart from cache misses.
- In contrast, dynamic loading (D) yields clear overhead compared to S—it represents the overhead of multiple versus single inheritance in a dynamic loading setting. Apart from AMD processors, this overhead is, however, slighter than between S and G.
- Summing both overheads makes the difference between global compilation and dynamic loading impressive—about 20%.

The differences are all the more significant that all measures include the time consumed by garbage collection, which is constant for all variants as it does not rely on any object-oriented mechanism. Now, the Boehm conservative garbage collector is certainly not optimized for the simple PRM object model—apart from the subobject-based variant—and a dedicated semi-conservative collector would reduce this common constant part and increase the relative differences.

processor			A-6	Athlon 64		A-7	Opteron Venus		I-8	Core2 T7200		I-9	Core2 E8500		Stripped executable on processor I-8		
frequency				2.2 GHz			2.4 GHz			2.0 GHz			3.16 GHz				
L2 cache			43.4s	1024 K		41.8s	1024 K		36.0s	4096 K		22.1s	6144 K		ref. size: 1220 KB		
year				2003			2005			2006			2008				
technique	scheme		AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
MC-BTD$_\infty$	CHA	G				-13.4	14.6	32.3	-14.0	7.3	24.8	-14.5	-2.0	14.6	15.7	21.6	5.1
MC-BTD$_2$	CHA	G	-13.7	5.0	21.6	-12.3	17.3	33.8	-12.7	6.9	22.4	-13.9	-2.6	13.1	-34.1	-30.5	5.5
MC-BTD$_\infty$	CHA	O				-1.6	23.7	25.7	1.9	13.9	11.8	2.4	15.2	12.5	-25.9	-22.0	5.3
MC-BTD$_2$	CHA	O	6.4	29.7	21.9	-0.4	23.9	24.5	0.1	12.7	12.6	1.8	13.8	11.7	-33.1	-29.5	5.4
MC		S	0	21.9	21.9	0	24.0	24.0	0	15.8	15.8	0	14.1	14.1	0	2.6	2.6
SO		D			–			–			–			–			–
IC		D	14.6	53.4	33.9	16.0	51.4	30.5	6.9	35.5	26.9	6.5	34.1	26.0	4.3	16.4	11.6
PH-and		D	18.1	73.1	46.6	17.3	71.5	46.2	8.0	46.4	35.6	6.9	45.2	35.8	16.7	39.3	19.4
PH-and+shift		D				21.5	85.5	52.6	13.5	66.4	46.6	15.2	66.4	44.5	25.2	50.8	20.4
PH-mod		D	110.2	345.8	112.1	96.9	300.5	103.3	25.5	158.5	105.9	25.0	119.8	75.8	16.7	35.7	20.4
PH-mod +CA		D				66.6	267.3	120.5	35.2	150.7	92.5	31.1	142.0	84.5	48.8	80.7	21.4

Last column presents statistics of stripped executable size on processor I-8, with the same conventions as for time measures.

Table 5. Compilation time according to implementation techniques and processors (cont.)

Global Optimization Levels (O and G). In contrast with the significant differences between compilation schemes, the differences between global optimization levels, e.g. type analysis algorithms or BTD depths, are too weak to draw firm conclusions. This is a consequence of statistics in Table 3 which show that the main improvement should come from monomorphic calls (BTD_0) which represent 64% of method calls. In contrast, BTD_1 and BTD_2 only amount to 30% of BTD_0 and the expected improvement would be less than proportional, because of mispredictions, hence hardly measurable on most processors. Finally, when $i > 2$, the number of BTD_i is too low to conclude whether BTD_i is an improvement on coloring or not.

With both G and O, the observations confirm this expectation. Therefore, this testbed is certainly unable to finely tune the optimal level of BTDs for a given processor and it is doubtful that any testbed could do it, since the optimal closely depends on the specific program type flows. So the decision must be drawn from theoretical considerations. BTD_1 should always be better than MC, since a mispredicted conditional branching has the same cost as an undirect branching. BTD_2 should likely be better than MC, since a single well-predicted branching makes it better. BTD_3 and BTD_4 probably represent the critical point. Overall, we present only the statistics for BTD_∞ and BTD_2. It would seem that BTD_∞ often improves on BTD_2 in G but not in O. This is consistent with the fact that BTDs are inlined in G, hence predictions are proper to a given call site, whereas they are shared in the thunks of O. Of course, sharing increases branching mispredictions. In contrast, the code is far smaller with sharing (Fig. 5).

Similar conclusions hold with type analysis algorithms. The difference in precision between algorithms is marked, but not enough to change the conclusions. For want of place, we do not present the statistics of polymorphism and runtime measures with RTA and CFA. The latter improves the monomorphic call rate by more than 10% and the gain of global compilation is increased in the same proportion, i.e. about 1-2%.

With G, the conclusion should be to use the best tradeoff between accuracy and compilation time, for instance with an explicit or implicit option that would allow the programmer to choose between various optimization levels. With O, the conclusion might be to use the simple CHA algorithm, which does not require any other data than *external models* and simplifies the overall architecture.

Dynamic Loading (D). The comparison between the different techniques compatible with dynamic loading mostly confirms previous theoretical analyses.

- When used for method invocation and subtype testing, PH-and yields very low overhead of about 4-8% on most processors—this is better than expected. It can be explained by the few extra `loads` from a memory area that is already used by the reference technique, hence without extra cache misses, plus a few 1-cycle instructions; the few extra cycles represent real overhead that is, however, slight in comparison with the overall method call cost.
- The extra instructions of PH-and+shift entails extra overhead, that is higher than expected since the extra instructions could have been done in parallel.
- Incremental coloring (IC) is close to PH-and, a little bit better but the difference is below the precision of measurement. In view of the respective load-time costs, PH-and should be preferred.
- In contrast, the overhead of PH-mod is much higher and highly variable, between 25 and 100%, when only used for method invocation and subtype tests.

Overall, PH-and is better than expected for method invocation and subtype testing, so it should provide very high efficiency in JAVA virtual machines for implementing interfaces. When used for attribute access, the overhead becomes less reasonable. In contrast, the integer division overhead is

higher than expected on many processors and it confirms that PH-`mod` should be reserved to processors that have efficient integer division—contrary to many processors tested here which use the floating-point unit for integer division.

Subobjects (SO). Statistics in Table 3 show that the pointer-adjustment count, augmented by the number of attribute accesses with `rst`≠`aic`, represents about 65% of the attribute access count. As accessor simulation represents a pointer adjustment, the overhead of subobjects, when restricted to attribute access and pointer adjustment, should be about 65% of the overhead of accessor simulation with method coloring. On most processors, this would be greater than the overhead of IC or PH-`and`. Moreover, we have not considered the overhead in method invocation resulting from pointer adjustment on the receiver. This explains why the overall overhead of subobjects is expected to be always higher than IC and PH-`and` with attribute coloring, though always lower than the overhead of the same techniques with accessor simulation.

Accessor Simulation (AS). In all cases, accessor simulation entails significant overhead compared to attribute coloring. It was of course expected—especially in view of the high number of attribute accesses (Table 3)—since accessor simulation replaces the single `load` of attribute coloring by a sequence similar to method invocation, apart from an actual function call. So, it adds extra accesses to memory areas that are possibly not in cache with attribute coloring—hence, it increases cache-miss risks. Moreover, the single load of attribute coloring can be more easily done in parallel than the several-instruction code sequence of accessor simulation.

For all techniques used with dynamic loading (D), accessor simulation provokes apparent non additive overhead, as a kind of inverted triangular inequality. For a given technique, say IC, the overhead IC-AS/MC-AC is far greater than the sum of IC-AC/MC-AC plus MC-AS/MC-AC. This is partly explained by the fact that the overhead IC-AC/MC-AC, which only concerns method invocation, must be extrapolated to attributes, hence multiplied by $\frac{4285+2510}{2510} \approx 2.6$, according to statistics in Table 3. So IC-AS/MC-AC must be compared to IC-AC/MC-AC (multiplied by 2.6) plus MC-AS/MC-AC. This explains most of the observed overhead. An explanation of the rest may be that the sequence forms a bottleneck that blocks instruction-level parallelism, whereas the single coloring `load` can be done in parallel in most cases. The overhead is specially marked with PH-`mod` which is quite unreasonable with accessor simulation.

Nevertheless, these results are not definitive, because the accessor simulation overhead has been overestimated in our tests—indeed, true accessors are also intensively used in the tested programs, in such a way that they add both overheads of accessor methods and simulation. So accessor methods should be implemented by direct access to the attribute, as with AC, at least when the method is generated by the compiler in S and D schemes. However, in view of the statistics of accessors in Table 3, the improvement should be slight.

Cache (CA). As aforementioned, the observed cache-hit rate is not as high as reported by (Palacz and Vitek 2003). So one must expect that caching can only improve PH-`mod` with AC—indeed, with AS the cache-hit rate is too low, and with PH-`and`, the underlying technique is too efficient. Our observations confirm this analysis and we only include CA with PH-`mod` in Tables 4 and 5, as caching markedly degrades all other techniques. On processors I-5, I-8 and I-9 which have rather efficient integer division, the cache yields a slight extra overhead, hence ruling out caching. On all other processors, the cache slightly improves PH-`mod` but PH-`and` remains far better. As the cache markedly increases the overall table and code size, the winner is clearly PH-`and`. Moreover, the slight degradation of caching with accessor simulation accords with cache-hit rates in Table 3. A specific cache for attribute access might be a way of improving cache hits, to the detriment of the overall table size.

Overall, this confirms that caching can only be a solution if: (i) the underlying technique is unefficient and (ii) the number of cachable entities is not too high, e.g. with JAVA interfaces.

Size of Executable. Although the PRM testbed is not optimized from the memory occupation standpoint, some conclusions can be drawn from the statistics presented in the last column of Table 5. Global compilation (G) or link-time optimizations (O) markedly reduces the executable size. Moreover, G could be far better with dead code elimination. However, one observes that BTD_∞ markedly increases the program size when BTD are inlined, as in G. However, BTDs are expected to be time-efficient only when they are inlined. So this is another argument for combining BTDs with coloring.

With dynamic loading, the statistics may be less reliable. First, IC involves dynamic reallocation that are not taken into account, here. Moreover, PH has not been implemented in the most efficient way from the space standpoint (Ducournau and Morandat 2009), so the space overhead is certainly overestimated. However, a firm conclusion is possible for PH-`and+shift`—this technique was designed for reducing the table size, but the statistics show that it markedly increases the code size. So, a likely definitive conclusion would be to rule out PH-`and+shift`, since it degrades both time and space.

Finally, caching proves to be over space-consuming when it is inlined, like all techniques considered here. So, caching should likely be reserved to non-inlined techniques, for instance the bytecode interpreter of virtual machines. An alternative would be to use it with shared thunks, like in O.

Processor Influence. The processor influence is also significant, even though it does not reverse the conclusions.

Most processors present similar behaviour, although several provide some specific exceptions that make them unique:

- On I-2, many tests do not differ from each other by more than 2-3%—as the processor is not specially steady, this allows only to conclude that these techniques do not essentially differ. M-3 presents a similar behavior on a smaller subset of techniques (e.g. IC and MC-AS). On all other processors, the corresponding differences are marked.
- On AMD processors (A-6 and A-7), most overheads (e.g. of perfect hashing or accessor simulation) are doubled.
- PH-`mod` is markedly more efficient on recent Pentium (I-5, I-8 and I-9).

These aberrations might be explained, either by some artefact in the experiment, or by some specific feature of the processor—for instance, A-6 is both an early 64-bit that might be specially inefficient and a remote computer that might be specially noisy. Processors are presented and numbered in the decreasing order of the reference duration, which is strongly correlated with the manufacturing time. It is however hard to find strong correlations between the observed overheads and time or overall performance.

6. Related Works, Conclusions and Prospects

Related Works. There have been many works on implementation and compilation of object-oriented languages and programs. The most important part has been made in a dynamic typing setting and applied to SMALLTALK, SELF or CECIL, and also to *multiple dispatch* in languages like CLOS, CECIL or DYLAN. Although the techniques considered here often originate from these dynamic typing studies—e.g. coloring or BTD, the latter being an improvement of *polymorphic inline caches* (Hölzle et al. 1991)—static typing makes them much more efficient. Besides implementation techniques, a lot of works have also been done for optimizing object-oriented programs. For instance, the Vortex compiler is dedicated to the assessment of various optimizations techniques for JAVA and CECIL programs (Grove and Chambers 2001). In the C++ context, the various implementations of pointer adjustments (VBTRs, thunks, etc.) have been compared (Sweeney and Burke 2003) and different approaches have been proposed for optimizing the generated code by *devirtualization* (Gil and Sweeney 1999; Eckel and Gil 2000). However, under the CWA, these optimizations are outclassed by coloring. JAVA and .NET gave also rise to a lot of studies about interface implementation (Alpern et al. 2001a,b; Click and Rose 2002; Palacz and Vitek 2003) and adaptive compilers (Arnold et al. 2005). Our testbed could not include the latter because of its incompatibility with dynamic loading. Regarding interface implementation, besides PH and IC, they all offer non-constant-time efficiency and are mostly based on caching and searching. Their scalability is doubtful but it was not possible to include them in our testbed for a fair comparison, since PRM does not distinguish between classes and interfaces.

Finally, the point with object-oriented implementation does not limit to method invocation, attribute access and subtype testing. A lot of little mechanisms are also implied—interested readers are referred to (Ducournau 2002) for a survey. A major efficiency concern is about genericity. The implementations of generics lie between two extremes (Odersky and Wadler 1997). In *heterogeneous* implementation, e.g. C++ templates, each instance of the parametrized class is separately compiled. In *homogeneous* implementation, e.g. JAVA 1.5, a single instance is compiled, after replacing each formal type by its bound (this is called *type erasure*). These two extremes present an interesting time-space efficiency tradeoff. Heterogeneous approach is markedly more time-efficient than the homogeneous one when the formal type is instantiated by a primitive type—in contrast, in such a situation, type erasure forces automatic *boxing* and *unboxing*. On the other hand, the code and method tables are duplicated for each instantiation whereas homogeneous implementations share the same code and method table for different instantiations. Intermediate policies still represent a research issue. PRM relies on an homogeneous implementation and heterogeneous or mixed implementations are a matter of future research.

Conclusions. In this article, we have presented the empirical results of systematic experiments of various implementation techniques and compilation schemes, on a variety of processors. To our knowledge, this is the first systematic experiment that compares such a variety of implementation techniques and compilation schemes, with *all other things being equal*—the latter point was a major challenge of this work. Although these tests were performed on an original language and compiler, they provide reliable assessment of the use of the considered techniques in the setting of production languages like C++, JAVA or EIFFEL. The results confirm that global compilation markedly improves the runtime efficiency, even when many optimizations are not considered like dead-code elimination. In this setting, the combination of coloring and BTDs certainly provides the highest efficiency. In contrast, dynamic loading always implies marked overhead, even in the restricted case of JAVA interfaces, i.e. when coupled with attribute coloring (AC).

More specifically, the tests provide an estimation of the difference of efficiency that must be expected between languages like EIFFEL and C++—though our testbed does not equitably account for the template heterogeneous implementation—that are closely related to their specific compilation scheme, at least when the latter is used in a fully reusable way. They also provide an empirical assessment of the overhead of such functional features as multiple inheritance and dynamic loading.

Another contribution is a first empirical assessment of a new technique, *perfect hashing*, which is the first known

technique that is both time-constant and space-linear in a general multiple inheritance and dynamic loading setting. The conclusions are two-fold. PH-`and` overhead is quite reasonable and makes the technique recommended for implementing JAVA interfaces, all the more so since recent research shows that its space occupation can also be very good (Ducournau and Morandat 2009). On the contrary, PH-`mod` is unreasonably inefficient on many processors. Finally, PH-`and+shift` is likely not justified—its slight gain in method tables does not offset the slight time overhead and code length increasing. Moreover, our tests show that caching cannot be a solution when the underlying technique is efficient enough.

In contrast, the conclusion concerning the mixed compilation scheme with link-time global optimization (O) is a little bit disappointing. A slight improvement was expected and the tests show instead a slight overhead on most processors. From the time standpoint, the link-time complication of these global optimizations might not be justified since the simple global linking (S) is functionally equivalent. However, this is only a first test. More complete optimizations, coupled with the hybrid scheme (H), should increase the time efficiency.

The tests were performed on a variety of processors, most of them with the same x86 architecture. Though most processors behave in a similar way, several exceptions lead us conclude that language implementors should offer alternative implementations that might be customized on each specific computer and operating system.

Of course, these experiments do not allow us to definitely decide for all processors and programs. The choice of an implementation will always depend on functional requirements such as dynamic loading. If dynamic loading is required, with full multiple inheritance, the C++ implementation likely represents the best choice from the time standpoint—the overhead of subobjects is counterbalanced by template heterogeneous implementation and the fact that primitives types are not integrated in the object type system. However, the scalability is doubtful from the space standpoint—the worst-case table size is cubic in the number of classes and compiler-generated fields in the object-layout can be over space-consuming. With JAVA interfaces, PH-`and` is certainly the best underlying implementation technique and adaptive compilers might focus on monomorphic calls. If dynamic loading is not required, the hybrid compilation scheme that combines separate compilation of libraries and global compilation of programs likely provides the best tradeoff between flexibility for rapid recompilations and efficiency of production runtimes. In this framework, the combination of coloring and BTDs provides the most compact and efficient code.

Prospects. Our experiments must be completed in several directions. Regarding the PRM testbed:

- For want of time and space, the presented statistics are not complete—compilation and link time, processor cache misses, runtime memory occupation should also be considered.
- The optimization of schemes O and G is not achieved, especially from a space standpoint; all method tables or object layouts are not optimized and the dead code is not eliminated—this would be easy in global compilation (G), more difficult with link-time optimization (O) as usual linkers are not equipped for deleting code.
- The techniques used in the PRM compiler are fully portable with respect to processors; however global optimizations involved in the O scheme closely depend on linkers and operating systems; so a general solution is required before using this scheme in a production compiler.
- Some techniques can still be optimized—for instance, accessor simulation. Generally, it should not be used with accessor methods. With global compilation (G), it should be optimized for taking possible invariance into account, in a way similar to monomorphic calls.
- Other compilation schemes like the hybrid one (H) that has been presented in Section 3.4 should also be tested.
- Polymorphic handling of primitives types is done in PRM through a mixin of *tagging*—for small integers, characters and boolean—and automatic *boxing* and *unboxing*, as in JAVA; the testbed should also consider a precise assessment of these techniques.
- An efficient implementation of generics goes midway between homogeneous and heterogeneous implementations—this is a matter of further research, not only of integration into the PRM testbed.
- Apart from subobjects which might justify a fully conservative collector, a dedicated semi-conservative garbage collection should reduce the overall time, thus increasing the relative overheads. This might reverse conclusions of the comparison between subobjects and perfect hashing.
- Testing processors from other architectures is mandatory—this is the condition for these techniques being widespread. The testbed should also consider other C compilers than `gcc`.

Several experiments with production virtual machines could also take advantage of the techniques presented in this article. First, the efficiency of perfect hashing for interface implementation should be confirmed by large-scale tests. Moreover, the *thunk*-based technique of link-time global optimization (O) could also apply to *adaptive compilers*. Instead of recompiling methods when load-time assumptions are invalidated by some subsequent class loading, only thunks would need recompilation. In view of the high rate of monomorphic calls and the overhead of all techniques compatible with dynamic loading, this would certainly be

an improvement for method invocation when the receiver is typed by an interface—maybe also when its is typed by a class. It could be tested in the PRM testbed by coupling PH with global optimizations (O) but, like for IC, this would not fully account for the recompilations required by adaptive compilers.

Finally, in the state space of object-oriented programming language design, there remains a blind spot—namely a language with full multiple inheritance, like C++ and EIFFEL, fully compatible with dynamic loading, like C++ and JAVA, with a clean integration of primitive types, like EIFFEL and JAVA. JAVA-like boxing and unboxing would degrade usual C++ subobject-based implementation and adaptive compiler techniques are likely less adapted to subobjects than to invariant-reference implementations. However, the best alternative that we can currently propose, PH-`and` with accessor simulation, is almost twice as slow as the most efficient implementation with global compilation. So there is room for further research.

References

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

B. Alpern, A. Cocchi, and D. Grove. Dynamic type checking in Jalapeño. In *Proc. USENIX JVM'01*, 2001b.

M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, Feb. 2005.

David F. Bacon, M. Wegman, and K. Zadeck. Rapid type analysis for C++. Technical report, IBM Thomas J. Watson Research Center, 1996.

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

H.-J. Boehm. Space-efficient conservative garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'93)*, ACM SIGPLAN Notices, 28(6), pages 197–206, 1993.

Dominique Boucher. GOld: a link-time optimizer for Scheme. In M. Felleisen, editor, *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, pages 1–12, 2000.

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

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

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

Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theor. Comput. Sci.*, 182(1-2):1–143, 1997. ISSN 0304-3975. .

J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'95)*, pages 93–102, 1995a.

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, 1995b.

R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*, pages 211–214. ACM Press, 1989.

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

R. Ducournau. Implementing statically typed object-oriented programming languages. Rapport de Recherche 02-174, LIRMM, Université Montpellier 2, 2002. (submitted to ACM *Comp. Surv.*; revised July 2005; Aug. 2008).

R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. Rapport de Recherche 06-001, LIRMM, Université Montpellier 2, 2006. (submitted to ACM *Trans. Program. Lang. Syst.*; rev. Aug. 2008).

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

R. Ducournau. *Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual*. Sema Group, Montrouge, France, 1991.

R. Ducournau and F. Morandat. More results on perfect hashing for implementing object-oriented languages. Rapport de Recherche 09-001, LIRMM, Université Montpellier 2, 2009. (submitted to ACM *Trans. Program. Lang. Syst.*).

R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. Rapport de Recherche 08-017, LIRMM, Université Montpellier 2, 2008. (submitted to Science of Computer Progamming).

N. Eckel and J. Gil. Empirical study of object-layout and optimization techniques. In E. Bertino, editor, *Proc. ECOOP'2000*, LNCS 1850, pages 394–421. Springer, 2000.

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

M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco (CA), USA, 1979.

J. Gil and P. Sweeney. Space and time-efficient memory layout for multiple inheritance. In *Proc. OOPSLA'99*, SIGPLAN Notices, 34(10), pages 256–275. ACM Press, 1999.

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

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

Samuel P. Harbinson. *Modula-3*. Prentice Hall, 1992.

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.

S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, New York, 1996.

B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

F. Morandat, R. Ducournau, and J. Privat. Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique. In B. Carré and O. Zendra, editors, *Actes LMO'2009*, page ? Cépaduès, 2009. (to appear).

Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer, 1993. ISBN 3-540-60062-0.

S. Muthukrishnan and M. Muller. Time and space efficient method lookup for object-oriented languages. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 42–51. ACM/SIAM, 1996.

A. Myers. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 124–139. ACM Press, 1995.

M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. POPL'97*, pages 146–159. ACM Press, 1997.

Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In L. Cardelli, editor, *Proc. ECOOP'2003*, LNCS 2743, pages 378–404. Springer, 2003.

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

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

Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

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

G.L. Steele. *Common Lisp, the Language*. Digital Press, second edition, 1990.

Peter F. Sweeney and Michael G. Burke. Quantifying and evaluating the space overhead for alternative C++ memory layouts. *Softw., Pract. Exper.*, 33(7):595–636, 2003.

S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy, editors. *Ada 2005 Reference Manual: Language and Standard Libraries*. LNCS 4348. Springer, 2006.

Frank Tip and Peter F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36(12):927–982, 2000.

J. Vitek, R.N. Horspool, and A. Krall. Efficient type inclusion tests. In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 142–157. ACM Press, 1997.

O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 125–141. ACM Press, 1997.

Y. Zibin and J. Gil. Two-dimensional bi-directional object layout. In L. Cardelli, editor, *Proc. ECOOP'2003*, LNCS 2743, pages 329–350. Springer, 2003. URL .