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 threefold tradeoff between runtime efficiency, expressiveness (multiple inheritance), and modularity, i.e. open-world assumption (OWA). Runtime efficiency is conditioned by both the *implementation technique* and *compilation scheme*. The former specifies the data structures that support method invocation, attribute access and subtype testing. The latter consists of the production line of an executable from the source code. Many implementation techniques have been proposed and several compilation schemes can be considered from fully global compilation under the closed-world assumption (CWA) to separate compilation with dynamic loading under the OWA, with midway solutions. This article reviews a significant subset of possible combinations and presents a systematic, empirical comparison of their respective efficiencies with *all other things being equal*. The testbed consists of the PRM compiler that has been designed for this purpose. The considered techniques include C++ subobjects, coloring, perfect hashing, binary tree dispatch and caching. A variety of processors were 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 specific languages like C++ and EIFFEL. They also confirm that perfect hashing should be considered for implementing JAVA and .NET interfaces.

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Language classifications—object-oriented languages, C++, 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, closed-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

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

In spite of its 30-year maturity, object-oriented programming still has a substantial efficiency drawback in the *multiple inheritance* context and it 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*. Indeed, object-oriented implementation concerns a few basic mechanisms that are invoked billions of times during a 1-minute program execution. Constant time is the only way to bound the worst-case behaviour. Space linearity ensures that the implementation will scale up gracefully with the program size; we shall see, however, that linearity must be understood in a slightly specific meaning. Finally, the basic mechanisms must be implemented by a short sequence of instructions that must be inlined. This general implementation issue is exemplified by the way the two most used languages, namely C++ and JAVA, that support both multiple inheritance and dynamic loading, do not meet these criteria. When the `virtual` keyword is used for inheritance, C++ provides a fully reusable implementation, based on subobjects, which however involves many 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 was, until very recently, no efficient subtype testing available for this implementation. JAVA provides multiple inheritance of interfaces only but, even in this restricted setting, current

[1] The effect of omitting the `virtual` keyword is discussed below.

interface implementations are generally not time-constant (see for instance [Alpern et al. 2001a]). The present research was motivated by the following observation. Though object-oriented technology is mature, the ever-increasing size of object-oriented class libraries and programs 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: *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 dynamic type of the object.

Single vs. Multiple Inheritance. 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 pointers to functions, 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. 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, some 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 in 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. Hence, there is a real need for efficient object implementation in the multiple inheritance and static typing context. The multiple inheritance requirement is less urgent in the dynamic typing context. 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. This article focuses on static typing and multiple inheritance.

Compilation Schemes. Besides the implementation techniques, which concern 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/linking*. 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 *closed-world assumption* (CWA), that is ensured by *global compilation*, allows both efficient implementations and various optimizations that partly offset the late binding overhead. This approach is exemplified by the GNU EIFFEL compiler [Zendra et al. 1997, Collin et al. 1997]. A variety of combinations fall 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, namely 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 a temporary CWA, which 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. 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.

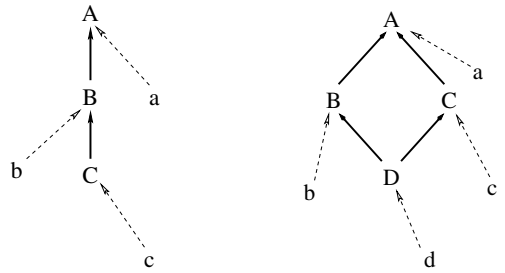
Languages. In principle, language specifications should be independent of implementation. However, in practice, many languages are closely dependent on a precise implementation technique or compilation scheme. For instance, the `virtual` keyword makes C++ inseparable from its subobject-based implementation [Ellis and Stroustrup 1990, Lippman 1996], whereas EIFFEL cannot be considered other than with global compilation, because of its unrestricted covariance which would make it unsafe and inefficient with separate compilation. Therefore, an objective comparison of the respective efficiencies of these languages is almost impossible because full specifications are not comparable. This article focuses on the core of object-oriented programming that is common to all languages. Therefore, the target languages can be thought of as mainstream languages like C++, JAVA, C# or EIFFEL. However, these languages only represent convenient well-known examples. The considered techniques should actually appear as universal techniques, that could apply to all languages, apart from general functional requirements like typing (static vs dynamic), inheritance (single vs multiple), compilation (separate vs global) and linking (static vs dynamic) that markedly hamper the implementation.

Contributions. Since the beginning of object-oriented programming, many implementation techniques have been proposed. Some of them are commonly used in production runtime 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 yield a fair comparison with alternative techniques, with *all other things being equal*. There are many reasons for 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 perfect reproducibility.

This article is thus a step in a project intended to produce fair assessments of various alternative implementation techniques, with *all other things being equal*. The previous steps included abstract analysis in Driesen's framework, and simulation of the memory occupation based on large-scale class hierarchies [Ducournau 2006, 2008, 2009]. In the past few years, we developed a new language, called PRM, and a compiler with an open modular architecture which makes it relatively easy to test alternative techniques. Early results presented empirical measures of program efficiency based on artificial micro-benchmarks [Privat and Ducournau 2005]. In this article, we present an empirical assessment of the time-efficiency of a real program on a variety of processors and according to the underlying implementation techniques and compilation schemes that are used to produce the executable. Our testbed involves meta-compiling. It consists of the PRM compiler, which compiles PRM source code to C code and is applied to itself. The tests consider the following techniques: (i) *coloring* [Ducournau 2006] which represents an 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) *perfect hashing* [Ducournau 2008] that has been recently proposed for JAVA interfaces under pure OWA, (iv) *incremental coloring* [Palacz and Vitek 2003] also proposed for JAVA interfaces, that is an incremental version of coloring which requires load-time recomputations, (v) *caching*, which amounts to memoizing the last access and might improve less efficient techniques. C++ *subobjects* are discussed but not tested because they have not yet been integrated in the testbed.

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, and 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, as these tests represent, to our knowledge, the first systematic comparisons between very different approaches with *all other things being equal*. Another contribution of this work is a careful analysis of the testbed to ensure that all other things are actually equal. Among other results, these tests give the first empirical assessment of (i) a recently proposed technique, perfect hashing; (ii) the overhead of OWA vs. CWA; (iii) the overhead of multiple vs. single inheritance; and (iv) a first step towards an empirical comparison between C++ and EIFFEL implementations.

Plan. 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 work, first conclusions and prospects. Preliminary results have been presented in French in [Morandat et al. 2009].



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

2. Implementation Techniques

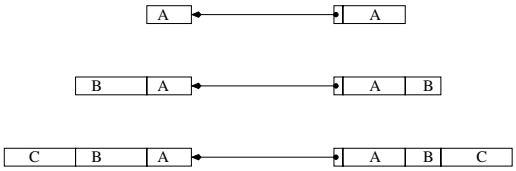
Implementation techniques are concerned with object representation, that is the object layout and the associated data structures that support method invocation, attribute access and subtype testing.

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 calls to pointers to functions 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; (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 accounts for method invocation and attribute access under the OWA. The efficacy of this implementation 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.

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, especially for avoiding bound checks.

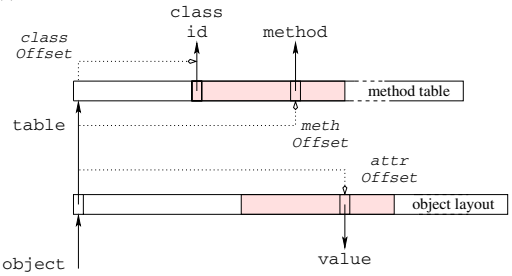
In this implementation, the total table size is roughly linear in the cardinality of the specialization relationship, i.e. linear in the number of pairs (x,y) such that x is a subtype (subclass) of y ($x \preceq y$). Cohen's display uses exactly one entry per such pair and the total table size is linear 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 $O(N+M)$ table size, but $O(\log\log N)$ invocation time, where N is the number of classes and M is the number of method definitions.



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.

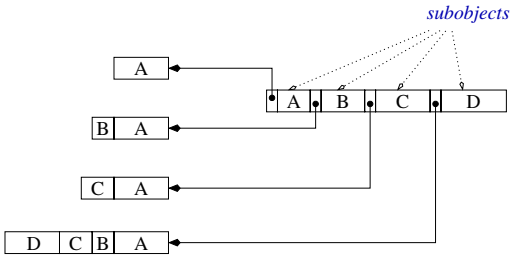
Figure 2. Single subtyping implementation

```
// attribute access
load [object + #attrOffset], value

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

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



The code sequences are expressed in the intuitive pseudo-code proposed by [Driesen 2001]. The diagram depicts the corresponding object representation. 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. Code sequences and object representation in single subtyping



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

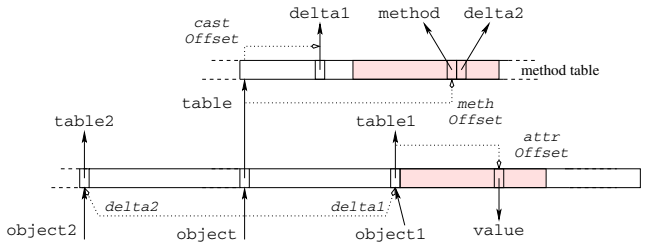
Figure 4. Subobject-based implementation

Notwithstanding efficiency considerations that are indeed central to this article, the three mechanisms that we consider (method invocation, attribute access and subtype testing) are equivalent in the extent that they are reducible to each other. Obviously, method tables are object layout *at the meta-level*. Hence, 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 calling `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, as 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). In the following, we will use this general equivalence in several ways.

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 instance, 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 introduced in *B* and *C*, thus prohibiting a sound implementation of *D*. Therefore, the 'standard' implementation (i.e. that of C++) of multiple inheritance in a static typing and separate compilation setting is based on *subobjects* (SO). The object layout is composed of several

```
// pointer adjustment
load [object + #tableOffset], table
load [table + #castOffset], delta1
add object, delta1, object1

// inherited attribute access
// lines 1-3
load [object1 + #attrOffset], value

// method invocation
load [object + #tableOffset], table
load [table + #methOffset+fieldLen], delta2
load [table + #methOffset], method
add object, delta2, object2
call method
```



The diagram depicts the precise object representation restricted to method invocation, attribute access and pointer adjustment. `object` is the current reference to the considered object. `delta1` is the pointer adjustment that is required to go 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 to go from `object` subobject to that of the class which defines the invoked method.

Figure 5. Code and object representation with subobjects

subobjects, one for each superclass of the object class. Each subobject contains attributes *introduced* by the corresponding superclass, together with a pointer to a method table which contains the methods *known* by the superclass (Fig. 4 and 5). Both invariants are dropped, as both reference and position depend now on the current static type. This is the C++ implementation, when the `virtual` keyword annotates each superclass [Ellis and Stroustrup 1990, Lippman 1996, Ducournau 2009]. It is time-constant and compatible with dynamic loading, but method tables are no longer space-linear. Indeed, the number of method tables is exactly the size of the specialization relationship. When a class is in single inheritance, its total table size is itself quadratic in the number of superclasses; hence, 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) 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, but they are quite numerous. They are also safe—i.e. the target type is always a supertype of the source type—and can be implemented more efficiently than subtyping tests.

Pointer adjustments can also be done with explicit pointers, called VBPTRs, in the object layout, instead of offsets in the method tables as in Fig. 5. Although VBPTRs are more time-efficient since they save an access to method table at each pointer adjustment, they are also over space-

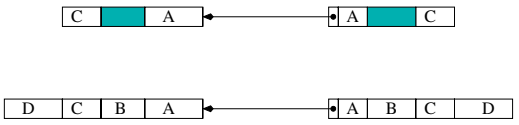
consuming. Therefore, we only consider here implementations that rely on offsets in the method table. They are also closest to most actual implementations.

Usually, pointer adjustments consist of upcasts, i.e. they involve an assignment from a subtype to a supertype. However, covariant return types, though type-safe, need downcasts. Indeed the type returned by the callee is a subtype of the static type in the caller, but the returned type is statically unknown in the caller, hence the adjustment must be done as if it were a downcast. 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 *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*. In our tests, we will thus supplement subobject-based implementations with perfect hashing (Section 2.5) for subtype testing.

Empty-subobject optimization (ESO) [Ducournau 2009] represents a further improvement that applies when a class does not introduce any attribute—hence the corresponding subobject is empty—especially when it does not 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, it is required in the PRM testbed for efficient *boxing* and *unboxing* of primitive types. Indeed, unlike C++ and like JAVA 1.5 and EIFFEL, the PRM type system considers that primitive types are subtypes of some general types like `Object` or `Any`. Unoptimized subobjects would yield heavy boxes.

Subobjects can also apply to JAVA interfaces, with an improved empty-subobject optimization that relies on class single inheritance. The technique, detailed in [Ducournau 2009] based on the bidirectional layout of [Myers 1995], is less expensive than general subobject-based implementations, though likely not space-linear. It would be interesting to test it but it is incompatible with our PRM testbed, because of the distinction between classes and interfaces, and with current JVMs, because it is not reference-invariant.

When the `virtual` keyword is not used—we call it *non-virtual* inheritance—C++ provides a markedly more efficient implementation with different semantics. In the diamond situation of Fig. 1, this semantics yields *repeated inheritance* of the diamond root. However, repeated inheritance is not a sensible feature because it might force the programmer to syntactically distinguish between an exponential number of interpretations—consider a chain of *n* diamonds. Hence, one must consider that non-virtual inheritance is not compatible with both multiple inheritance and OWA. As coloring is certainly more efficient than non-virtual implementation under the CWA, we only consider C++, in the following, under the *virtual* implementation and the OWA.



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. Hence, the implementation of *C* leaves a gap (in grey) for *B* in anticipation of *D*. The code sequences and object representation are the same as in Fig. 3.

Figure 6. Coloring 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] for method invocation, under the name of *selector coloring*. [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 the overall tractability. 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.

[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.

```
load [object + #id0ffset], id
comp id, id0
bgt #branch1
comp id id1
bgt #branch2
call #method1
jump #end
branch2:
call #method2
jump #end
branch1:
comp id id2
bgt #branch3
call #method4
jump #end
branch3:
call #method3
end:
```

Figure 7. Binary tree dispatch (BTD_2)

A detailed presentation of coloring is beyond the scope of this paper and readers are referred to [Ducournau 2006] for a review of 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 VFT. 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 the 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, thus 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 BTD relies on the conditional branching prediction of modern processors. Thanks to their pipe-line architecture, well-predicted branchings are free. On the contrary, mispredictions break the pipe and cost about 10 cycles or more, and most undirect branches are mispredicted—this misprediction cost thus holds for all VFT-based techniques. Readers are referred to [Driesen 2001] for a more in-depth analysis. An overall consequence is that BTD is statistically more efficient than VFT when the number of tests is small. It depends, however, on the statistical distribution of dynamic types on each call site, and it is easy to construct worst-case artificial programs whereby all predictions fail, making VFT far better than BTD. In the following, BTD_i will denote BTD of depth bounded by i. BTD_0 corresponds to static calls and BTD_∞ denotes unbounded BTD. Figure 7 depicts a dispatch tree of depth 2.

Overall, BTD is efficient when the number of expected types and competing methods is low—the corresponding call sites are then called *oligomorphic*—but coloring should be preferred when this number is higher, for *megamorphic* call sites. An interesting tradeoff involves combining BTD_k and coloring, with k not greater than 3 or 4. This makes the resulting technique time-constant and inlinable. Furthermore, method tables are restricted to the subset of methods that have a megamorphic call site. BTD also applies to subtype testing and attribute access but, in the context of global compilation, coloring is likely better.

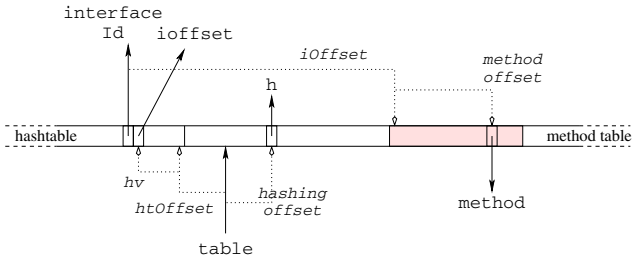
2.5 Perfect Hashing (PH)

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. Hence, $c \preceq d$ iff $id_d \in I_c$. This set I_c is immutable, hence it can be hashed with some *perfect hashing function* h_c, i.e. 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 *hash* functions were considered, namely *modulus* (noted `mod`) and bitwise `and`[3]. The corresponding techniques are denoted hereafter PH-`mod` and PH-`and`. However, these two functions involve a time/space efficiency tradeoff. The former yields more compact tables but the integer division latency may

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

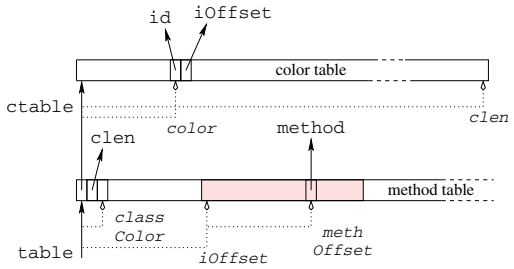
```
// preamble
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #interfaceId, h, hv
mul hv, #2*fieldLen, hv
sub table, hv, htable

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

// subtype testing
load [htable +#htOffset-fieldLen], id
comp #interfaceId, id
bne #fail
// succeed
```



The preamble is common to both mechanisms. The grey rectangle denotes the group of methods introduced by the considered interface.

Figure 8. Perfect hashing for JAVA interfaces

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

// method invocation
load [entry + #2], iOffset
add table, iOffset, methgr
load [methgr + #methOffset], method
call method

// subtype test
load [table + #clenOffset], clen
comp clen, color
ble #fail
load [entry], id
comp id, #targetId
bne #fail
// succeed
```



The preamble is common to both mechanisms. The implementation resembles PH, apart from the fact that the interface position results from a `load` instead of being the result of specific hashing. The location of the color can be the method table of the target class that is not represented here. The `loads` concern different memory areas and they can yield 3 cache misses. Moreover, the recomputable color table requires an extra indirection, together with its size (`clen`) and bound checking for subtype testing, and the color itself requires memory access (not represented in the diagram).

Figure 9. Incremental coloring for JAVA interfaces

be more than 20 cycles, whereas the latter takes 1 cycle but yields larger tables.

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 8 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. In a forthcoming paper [Ducournau and Morandat 2009], we improve the technique in several directions, especially with a new hashing function that combines bitwise `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 that allows for both multiple inheritance and dynamic loading at reasonable spatial cost and applies to both method invocation and subtype testing. Subobject-based implementation has the same properties but applies only to method invocation. Moreover, the space requirement of perfect hashing is far lower than that of subobjects.

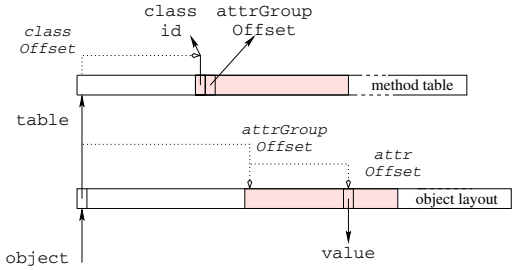
2.6 Incremental Coloring (IC)

An incremental version of coloring (denoted IC) has been proposed by [Palacz and Vitek 2003] for implementing interface subtype testing in JAVA. For the sake of comparison, an application to method invocation in the same style as for PH has been proposed in [Ducournau 2008]. As coloring needs the CWA, IC can require some load-time recomputations. Hence, its data structures involve extra indirections and several unrelated memory locations that should increase cache misses (Figure 9). Readers are referred to [Ducournau 2008] for a discussion on implementation details.

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

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



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 10. Accessor simulation with method coloring

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

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

Figure 11. Separate caches in method tables

an accessor. Then the attribute layout of a class does not have to be the same as the attribute layout of its superclass. A class will redefine 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 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. 10) [Myers 1995, Ducournau 2009]. With 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 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 be used 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 the object layout are considered to be over space-consuming.

2.8 Caching and Searching (CA)

An implementation policy that is often applied to dynamic typing or JAVA interfaces involves coupling some implementation technique (that is expected, here, to be rather naive and inefficient) with caching for memoizing the result of the last search. For instance, with JAVA interfaces, the underlying technique could be PH or IC and the method table would cache the interface ID and the group offset of the last succeeding access [Alpern et al. 2001a,b, Click and Rose 2002]. Of course this cache might be used 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. Finally, the cache may be common to all three mechanisms, or specific to each of them. Obviously, the improvement is a matter of statistics and those presented in [Palacz and Vitek 2003] for subtype testing show that, according to the different benchmarks, cache miss rates can be as low as 0.1% or more than 50%. Fig. 11 presents the code sequences of caching for method invocation and subtype testing—they are markedly longer than for the original implementations. Moreover, the worst case remains rather inefficient and the best case is hardly better than PH-`and` for method invocation and identical to Cohen's display for subtype testing. Like IC and unlike all other techniques, caching also requires method tables to be writable, hence allocated in data memory segments.

Cache-hit rates can be further improved with multiple caches (CA_n). For instance, with n caches, classes (or interfaces) are statically partitioned into n sets, for instance by hashing their name. In the method tables, the cache data structure (i.e. the offsets `#cacheTypeId_i`, `#cacheMethId_i` and `#cacheMethOffset_i`) is then replicated n times. Fi-

nally the code of each call site is that of Fig. 11, with the cache structure, i.e. the index i, corresponding to the `#targetId` of the given site. Hence the cache miss rate should asymptotically tend towards 0 as n increases—however, the best and worst cases are not improved. In this approach, the tables of the underlying implementation are only required to contain class or interface IDs for which there is a collision on their proper cache.

In our tests, we will also consider PH when it is coupled with caching. One might expect, for instance, that caching degrades PH-`and` but improves PH-`mod`.

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 already mentioned, 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 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 of efficiently implementing 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.

When only (i) holds, a simple *class hierarchy analysis* (CHA) [Dean et al. 1995b] provides a rough approximation of the call graph that is sufficient for identifying many monomorphic call sites. With global compilation (scheme denoted G), when the program entry point is known (point (iii)) and the language does not provide any metaprogramming facility, a more sophisticated *type analysis* precisely approximates the receiver *concrete type* at each call site, thus making it easy to identify mono-, oligo- and mega-morphic sites, so that each category can be implemented with the best technique, respectively, static calls, BTD_i and coloring. 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, giving up 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) above), 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 collects the models of all classes and colors all of the entities, before substituting values to the different symbols, as a linker commonly does. The linker also generates method tables.

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, which 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 with the name of the considered method and the static type of the receiver. Then, at link-time, a stub function—called a *thunk* as 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.

[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).

Implementation Technique		Compilation Scheme				
		D dynamic	S separate	O optimized	H hybrid	G global
subobjects	SO	◇	◇	∗	∗	∗
perfect hashing	PH	○	●	∗	∗	∗
incremental coloring	IC	○	●	∗	∗	∗
caching	CA	○	●	∗	∗	∗
method coloring	MC	⋆	●	●	◇	●
binary tree dispatch	BTD	⋆	⋆	●	◇	●
attribute coloring	AC	⋆	●	●	◇	●
accessor simulation	AS	○	●	●	◇	●

●: Tested, ○: Extrapolated, ◇: Not yet tested, ∗: Non-interesting, ⋆: Incompatible

Table 1. Compatibility between compilation schemes and implementation techniques

	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

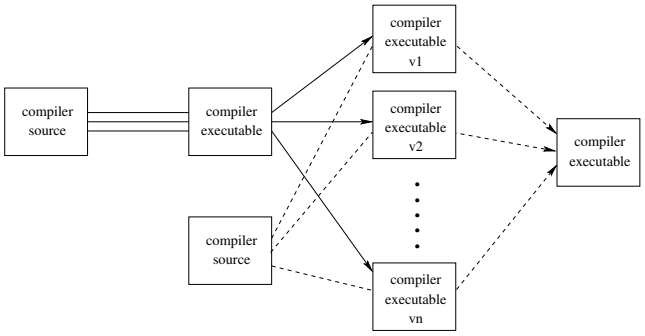
Table 2. Expected efficiency

An 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—remember that our tests consider only run-time efficiency.

Not all compatible combinations are interesting to test. For instance, all techniques that are compatible with the OWA are less efficient than coloring and BTD. Testing them in O, H and G schemes would thus be wasting time. Moreover, for these techniques, there is no practical difference between D and S in our testbed because it cannot reproduce the memory-allocation effects of dynamic loading. Hence, S is the right scheme for comparing the efficiency of implementation techniques like SO, PH, IC and MC. O, H and G are the right ones for comparing MC and BTD. Moreover, with O and G, the comparison can also consider various type analysis algorithms (CHA, RTA or CFA) and polymorphism degrees for BTD. 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 as it 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. Overall, several tens of combinations can be tested and we present only the most significant.



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 12. The PRM testbed

4. Compilation Testbed

These experiments are original as they compare different implementation techniques in a common framework that allows for a fair comparison with *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]. The benchmark program is the compiler itself, which is written in PRM and compiles the PRM source code into C code. There are many compilers in the picture, so Figure 12 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. This 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 this case, all PRM link-time optimizations are deactivated. Hence, although these tests rep-

number of		static	dynamic
class	introductions	532	—
	instantiations	6651	35 M
	subtype tests	194	87 M
method	introductions	2599	—
	definitions	4446	—
	calls	15873	1707 M
BTD	0	124875	1134 M
	1	848	61 M
	2	600	180 M
	3	704	26 M
	4..7	1044	306 M
	8	32	228 K
attribute	introductions	614	—
	accesses	4438	2560 M
	`rst=aic`	2996	1903 M
	accessor	361	229 M
pointer	upcasts	9133	783 M
adjustments	downcasts	3254	1393 M

The "static" column depicts the number of program elements (classes, methods and attributes) and the number of sites for each mechanism. The "dynamic" column presents the invocation count at run-time (in millions or thousands). Method call sites are separately counted according to their polymorphism degree with CHA, 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). Upcasts 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. Downcasts consist of covariant return types.

Table 3. Characteristics of the tested program

resent a kind of simulation, they must provide reliable extrapolation. Only the effect of cache misses is likely underestimated, especially for incremental coloring. Here all color tables are 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. 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 its branch number is between $(2^{i-1}+1)$ and 2^i.

Finally, the cache-hit rate has been measured (Table 4) when coupling perfect hashing (PH) with caching (CA). Of course, it does not depend on the specific technique that is associated with CA. Cache hits and monomorphic calls represent dual data. Monomorphism is a permanent link-time feature of a call site that always calls the same method, whereas a cache hit is a momentary run-time characteristics of a method table that is used for an access to some data that is introduced by the same superclass as in the previous access. With attribute coloring (AC), the measured cache-hit rate is between 60 and 80% according to whether the cache is common or separate, single or multiple. As expected, multiple separate caches have better efficiency. The

cache	with AC		with AS	
number	separate	common	separate	common
1	68	63	62	39
2	71		64	
4	79		67	

The cache-hit rate is presented (in percentage), with attribute coloring or accessor simulation, with separate caches or a common cache for all mechanisms, and according to the number of caches.

Table 4. Method table cache-hit rate

effect of separate caches is more marked with accessor simulation (AS), with the cache-hit rate increasing from less than 40% to more than 60%. With separate caches, the observed cache-hit rates are about the average of those reported by [Palacz and Vitek 2003]. Multiple caches are needed because the cache is used here for all classes, whereas its use was restricted, in the aforementioned paper, to interfaces.

The dynamic statistics from Tables 3 and 4 have been used for selecting variants that would be interesting to measure. Indeed, it would be vain to compare the execution times of variants that present very close statistics.

Test Significance. 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 large and fully object-oriented program intensively uses the basic mechanisms that are tested, namely a total of more than 4 billions invocations of all three mechanisms. Moreover, as the experiments compare two implementations with all other things being equal, the sign of the differences should hold for most programs and only the order of magnitude should vary, at least when the difference is not close to 0 and when the comparison focuses on a single parameter, i.e. in the same row or column in Tables 5 and 6. Our argument for supporting this claim is that the differences only concern the tested code sequences and the related cache misses. All other things are equal. Moreover the behaviours of all implementations are very similar from the cache standpoint, apart from BTD, since it does not access method tables, and IC, because it accesses extra memory areas. For the former, both considerations act in the same direction, i.e. bounded BTD are intrinsically faster with less cache misses. For the latter, the testbed is unable to take all load-time effects into account, hence it underestimates the cost of cache misses. IC is thus likely the only exception to our claim and we will take it into account in our conclusions.

This single-benchmark 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 compensation 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 to draw any firm conclusions. In the following, we consider relative differences, of the form $(test - ref)/ref$. Of course, if the numerator is small and the denominator is overestimated, for instance if the PRM compiler was one order of magnitude slower than it could be, the results might be meaningless. Therefore, we must also convince readers 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*. The GNU EIFFEL compiler, SMART EIFFEL, represents, however, a convenient reference because it involves a similar meta-compilation framework. It uses global compilation (G) and is considered very efficient—see Section 2.4. Both languages are fully-fledged object-oriented languages that provide similar features, and both compilers have close characteristics such as type analysis and the same target language. Thus we compared the compilation time of both compilers, from the source language (EIFFEL or PRM) to C. The compilation times were quite similar, about 60 seconds on the considered processor. Although this does not mean that the PRM compiler is as efficient as SMART EIFFEL, it is however a strong indication that it 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. This is even 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. Thus two compiler variants differ only by the code sequences of the considered techniques, with 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. 12) is not only a metaphor. Incidentally, this determinism ensures that the compiler bootstrap represents an actual fixpoint. All claimed 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[5].

[5] In early tests, compilation was not deterministic and there were marked variations in execution times between several generations of the same variant. Hence, the variation between different variants was both marked and meaningless.

Therefore, in principle, the statistics shown in Table 3 should not depend on compilation variants, though some of them interest only some specific variants. However, in spite of the compilation determinism, a compiled program is not exactly deterministic for these fine-grained statistics. Indeed, hashing object addresses is inherently not deterministic. 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. Above all, it does not modify the program logic because all hash structures used by the PRM compiler are order-invariant—that is all iterations follow the input order.

Processors. The tests were performed on a variety of processors (Tables 5 and 6):

- I-2, I-4, I-5, I-8 and I-9, from the Intel® Pentium™ family;
- 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;
- P-3 is a PowerPC G5, designed by IBM® and Apple®, under Mac OS X 10.5.3, with `gcc` 4.0.1.

Non-Linux linkers presented technical drawbacks that currently hinder global optimizations (O) on processors S-1 and P-3. All tests use Boehm's garbage collection [Boehm 1993] and the test on processor I-8 has also been run without GC, for the sake of comparison. 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. This has been done for some processors (e.g. I-2, I-4, I-8) but 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. Thus the peak speed can be very high but only for a short duration. Inserting a pause between each two runs seemed to fix the point and I-8 now provides one of the most steady testbeds.

Overall, we assume that the difference between two runs of the same executable is pure noise and this noise does not depend on the specific variant, but only on the test time and duration. As the noise is not symmetrical—it is always positive—we took, for each measure, the minimum value among several tens of runs.

processor		S-1	UltraSPARC III		I-2	Xeon Prestonia		P-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		123.2s	8192 K		87.4s	512 K		62.3s	512 K		43.3s	2048 K		34.8s	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$_\infty$ RTA	G	-22.6	-11.5	14.4	-10.9	-2.2	9.7	***	***	***	-13.3	-1.0	14.1	-8.9	2.9	13.0
MC-BTD$_2$ RTA	G	-22.2	-10.9	14.6	-11.7	-3.8	10.6	-28.4	-13.0	21.5	-13.7	-1.3	14.4	-10.2	-0.8	10.5
MC-BTD$_\infty$ CHA	O	***	***	***	-2.9	1.4	4.5	***	***	***	-3.0	4.9	8.2	2.7	19.4	16.2
MC-BTD$_2$ CHA	O	***	***	***	-5.4	-2.8	2.8	***	***	***	-5.9	2.3	8.7	-2.7	14.2	17.3
MC	S	0	9.8	9.8	0	5.7	5.7	0	18.2	18.2	0	7.9	7.9	0	11.1	11.1
IC	D	13.7	34.1	17.9	5.3	14.5	8.7	13.4	27.2	12.1	4.3	16.6	11.8	7.7	28.5	19.2
PH-`and`	D	13.4	35.4	19.5	2.5	14.5	11.6	8.1	24.9	15.6	4.2	19.0	14.2	6.2	31.4	23.8
PH-`and+shift`	D	14.7	38.0	20.3	10.6	25.5	13.6	13.4	35.5	18.9	6.9	28.7	20.4	10.3	45.3	31.8
PH-`mod`	D	81.1	226.0	80.0	28.6	104.3	58.8	49.1	146.1	65.1	55.2	172.0	75.2	24.4	106.3	65.8
PH-`mod+CA`$_4$	D	33.1	121.0	66.1	21.1	81.2	49.6	24.7	87.5	50.3	28.1	98.2	54.7	21.4	82.7	50.5

Each subtable presents the results for a precise processor, with the processor characteristics and the reference execution time. All other numbers are percentage. Each row describes a method invocation and subtype testing technique. For all 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. On P-3, BTD$_2$ is replaced by BTD$_0$.

Table 5. Execution time according to implementation techniques and processors

5. Results and Discussion

Tables 5 and 6 present, for each tested variant and processor, the time measurement and overhead with respect to the full coloring implementation (MC/AC). Overall, notwithstanding some exceptions that will be discussed hereafter, these tests exhibit many regularities.

Difference Magnitude. Before discussing measures, it is important to correctly interpret the overhead magnitude. The differences are all the more significant since all measures include the time consumed by garbage collection (GC), which is roughly the same for all variants as it does not rely on any object-oriented mechanism. In order to provide an estimation of the collector cost, the last column of Table 6 presents the same test with a deactivated GC, on processor I-8. It turns out that garbage collection takes between 14.5 and 15 seconds with all variants but MC-BTD-O, i.e. almost 50% of the reference time. Hence, all overheads are roughly doubled. Strangely enough, with MC-BTD-O, garbage collection takes only 13.4 seconds. We cannot explain this difference. Of course, the GC overhead depends on executable and memory sizes. This test was performed on a computer equipped with 2 G-bytes of memory, which is sufficient for running all variants without swapping. These measures must not be used to deny the usefulness of garbage collection. On a real-life computer, the tested program should share memory with many other programs and 2 G-bytes could not be dedicated to it. Moreover, Boehm's collector is conservative and not optimized for the simple object representations that are tested. A type accurate collector would be markedly more efficient [Jones and Lins 1996].

Overall, below 1%, a difference is likely meaningless, hence decimal digits must be handled with care. In contrast, 10% represents a marked difference, since considering the object-oriented part doubles the overhead. Finally, 50% overhead is dramatic.

Compilation Schemes.

- As expected, global compilation (G) was found to be markedly better than separate compilation (S). The high ratio of monomorphic calls 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) provides only a small improvement. This means that the gain resulting from 64% of static calls is almost 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.
- 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 G and D impressive, about 15% with multiple subtyping (AC) and 40% with full multiple inheritance (AS).

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 allow us to draw firm conclusions. This is a consequence of the 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 20% 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

processor		A-6	Athlon 64		A-7	Opteron Venus		I-8	Core2 T7200		I-9	Core2 E8500		I-8	without GC	
frequency			2.2 GHz			2.4 GHz			2.0 GHz			3.16 GHz				
L2 cache		34.0s	1024 K		32.8s	1024 K		30.4s	4096 K		18.5s	6144 K		15.7s	2 G-bytes	
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$_\infty$ RTA	G	-12.5	2.9	17.6	-13.7	9.9	27.3	-9.4	7.5	18.6	-10.3	0.6	12.1	-19.2	12.7	39.5
MC-BTD$_2$ RTA	G	-12.5	1.1	15.5	-13.1	10.7	27.3	-9.7	7.0	18.5	-9.7	0.6	11.5	-18.5	12.9	38.6
MC-BTD$_\infty$ CHA	O	5.8	27.8	20.8	4.0	23.3	18.6	1.1	12.1	10.8	1.8	15.0	12.9	10.2	31.0	18.9
MC-BTD$_2$ CHA	O	3.4	27.3	23.1	2.0	22.8	20.4	-1.8	10.1	12.2	-2.5	13.0	16.0	5.4	27.9	21.3
MC	S	0	15.8	15.8	0	15.2	15.2	0	11.3	11.3	0	10.3	10.3	0	21.6	21.6
IC	D	14.9	40.1	21.9	12.6	38.9	23.3	7.7	29.7	20.5	8.0	29.7	20.1	14.8	56.5	36.4
PH-`and`	D	15.2	52.4	32.2	16.8	57.3	34.7	5.1	30.0	23.7	6.2	30.1	22.6	9.6	55.9	42.2
PH-`and+shift`	D	19.5	64.4	37.6	18.8	61.5	36.0	7.8	43.6	33.2	8.3	43.3	32.3	14.1	81.9	59.4
PH-`mod`	D	80.1	235.4	86.2	73.4	221.5	85.4	19.5	108.7	74.7	17.6	85.7	57.9	35.2	209.1	128.6
PH-`mod`+CA$_4$	D	40.4	141.2	71.8	38.3	129.3	65.8	19.6	81.3	51.6	20.8	74.6	44.5	37.4	155.7	86.1

Last column presents measures on processor I-8 without garbage collection.

Table 6. Execution time according to implementation techniques and processors (cont.)

BTD$_i$ is an improvement on coloring or not. Thus we present only the statistics for BTD$_\infty$ and BTD$_2$.

With both G and O, the observations confirm this expectation. Therefore, this testbed is certainly unable to finely tune the optimal level of BTD for a given processor and it is doubtful that any testbed could do it, since the optimal closely depends on the specific type-flow of programs. Thus the decision must be drawn from abstract 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. It would seem that BTD$_\infty$ often improves on BTD$_2$ in G but not in O. This is consistent with the fact that dispatch trees are inlined in G, hence predictions are proper to a given call site, whereas they are shared in the thunks of O. Indeed, sharing increases branching mispredictions. In contrast, the code is far smaller with sharing (Fig. 7). Of course, this discussion should be reconsidered with processors without branching prediction, e.g. on small embedded systems.

Similar conclusions hold with type analysis algorithms. As CHA, in spite of its simplicity, gives very good results, more accurate approximations cannot change the conclusions. Hence, we do not present the statistics of polymorphism and run-time measures with RTA and CFA.

With G, the solution would 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 solution 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 3-8% on most processors. This could be explained by the few extra `load`s from a memory area that is already used by the reference technique, hence without extra cache misses, plus a few 1-cycle instructions; these extra cycles represent real overhead that is, however, slight in comparison with the overall method call cost. 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`, but not better. With accessor simulation, the difference is below the measurement precision. In contrast, the overhead of PH-`mod` is much higher and highly variable, between 17 and 80%, when only used for method invocation and subtype tests.

Overall, PH-`and` outclasses all considered alternatives for method invocation and subtype testing from the time standpoint. It is also better than PH-`and+shift` and not worse than PH-`mod` from the space standpoint (Fig. 7). In view of the respective load-time costs and of the underestimation of IC cache misses, PH-`and` should also be preferred over IC. This complete win is rather unexpected and PH-`and` 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 for processors that have very efficient integer division, contrary to many processors tested here which use the floating-point unit for integer division.

Cache (CA). As mentioned above, the observed cache-hit rate is highly variable according to the cache configuration, and the time measurements corroborate the statistics from Table 4. We tested caches with PH-`and` and PH-`mod`. As expected, caching degrades PH-`and` on all processors and with all cache configurations. Moreover, only multiple separate caches improve PH-`mod` on some processors. Therefore, we only present 4-fold separate caches (CA$_4$) with PH-`mod`.

On all processors, the cache slightly improves PH-`mod` with accessor simulation. In contrast, on I-8 and I-9 proces-

sors which have rather efficient integer division, the cache yields slight extra overhead with attribute coloring. 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`.

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

Accessor Simulation (AS). In all cases, accessor simulation entails significant overhead compared to attribute coloring. This 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. Hence, it adds extra access to memory areas that are possibly not in cache with attribute coloring, and it increases cache-miss risks. Moreover, the single load of attribute coloring can be more easily done in parallel than the several-instruction 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 difference between IC-AS and MC-AC is far greater than the sum of differences on the same row and column. This is explained by the fact that the first difference only concerns method invocation. Hence, it must be extrapolated to attributes by multiplying it by $\frac{2560+1707}{1707} \approx 2.5$, according to statistics in Table 3. Therefore, IC-AS/MC-AC must be compared to IC-AC/MC-AC (multiplied by 2.5) plus MC-AS/MC-AC. This explains the observed overhead for IC, PH-`and` or even PH-`mod`, in spite of the dramatic magnitude of the latter. From a methodological standpoint, it shows that cautious extrapolation is possible from tested techniques to a non-tested one. We shall apply this idea to subobjects just below.

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. Hence, 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. In G, accessor simulation should be used only on the attributes whose position vary according to classes. The resulting improvement should be as important as with monomorphic calls.

Subobjects (SO). We could not achieve on time subobject-based implementation. This is indeed the most demanding as pointer adjustments do not pardon any error. However, accessor simulation (AS) involves pointer adjustment and represents a convenient lower bound of the overhead yielded by subobjects. Indeed, an upcast adjustment is equivalent to AS with method coloring, whereas a downcast adjustment is roughly equivalent to AS with PH-`and`, since PH is used for subtype testing with subobjects. Statistics in Table 3 show that the upcast count (783), augmented by the number of attribute accesses with `rst`$\neq$`aic` (657), represents more than 50% of the attribute access count (2560). The downcast count (1393) too is more than 50% of the attribute access count. Therefore, the overhead of subobjects, when restricted to attribute access and pointer adjustment, should be at least 50% of the difference between AS and AC with method coloring (MC-S) plus 50% of the same difference with PH-`and`. Overall, without taking receiver adjustment into account, the overhead of subobjects would be about 20% on processor I-8. Hence, the total overhead of subobjects is expected to be markedly higher than IC and PH-`and` with attribute coloring, though likely always lower than the overhead of the same techniques with accessor simulation. This confirms that all implementations that are compatible with dynamic loading are costly. There remains, however, to precisely compare SO to PH-`and`. With attribute coloring, this would provide an estimation of the overhead of full multiple inheritance w.r.t. multiple subtyping under OWA. With accessor simulation, subobjects should be more efficient, but this requires an experimental confirmation.

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. For instance, A-7 is the only processor for which IC is markedly better than PH-`and`. On all non-Intel processors like S-1, P-3, A-6 and A-7, the magnitude of most differences is almost doubled in comparison to Intel processors. A particular case is PH-`mod`, which is markedly more efficient on recent Pentiums (I-8 and I-9) and dramatically inefficient on some other processors (S-1, I-4, A-6, A-7). These variations might be explained, either by some artefact in the experiment, or by some specific feature of the processor. The sample is however too small to draw any conclusion about processor families. 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 close correlations between the observed overheads and time or overall performance.

Size of Executable. Table 7 presents similar statistics of executable size instead of duration. Although the PRM testbed is not optimized from the memory occupation standpoint, some conclusions can be drawn from these statistics. Global compilation (G) and link-time optimizations (O) markedly reduce the executable size. G is also improved by dead code elimination, but the PRM modular architecture makes this improvement slight. In contrast, BTD_∞ markedly increase the program size when dispatch trees are specific to each site, as in G. However, BTD is expected to be time-

			Stripped executable on processor I-5 ref. size: 914 KB		
technique	scheme		AC	AS	AS/AC
MC-BTD$_\infty$	RTA	G	22.0	30.9	7.3
MC-BTD$_2$	RTA	G	-26.9	-17.0	13.5
MC-BTD$_\infty$	CHA	O	-5.1	-0.1	5.2
MC-BTD$_2$	CHA	O	-14.9	-10.4	5.3
MC		S	0	11.7	11.7
IC		D	19.0	33.3	11.9
PH-`and`		D	24.5	45.5	16.9
PH-`and+shift`		D	37.0	61.6	18.0
PH-`mod`		D	25.9	41.6	12.5
PH-`mod+CA`$_4$		D	75.6	104.7	16.7

In K-bytes and percentage. Conventions are the same as for time measures.

Table 7. Size of stripped executable on processor I-8

efficient only with proper trees. So space is another argument in favor of combining BTD with coloring.

With dynamic loading, the space statistics may be less reliable. First, IC involves dynamic reallocations 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], and the space overhead is certainly overestimated. However, a firm conclusion is possible for PH-`and+shift`. Indeed, this technique was designed to reduce the table size, but the statistics show that it markedly increases the code size. Hence, a definitive conclusion would be to rule out PH-`and+shift`, since it degrades both time and space. The difference between PH-`mod` and PH-`and` is not marked. This is apparently contradictory with the aforementioned time/space tradeoff [Ducournau 2008]. Actually, this tradeoff would be obtained with class hierarchies larger by an order of magnitude, but the space increase is likely low in comparison with the overall size.

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

6. Related Work, Conclusions and Prospects

Related Work. There has been a lot of work on implementation and compilation of object-oriented languages and programs. The most important part was 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, with the latter being an improvement of *polymorphic inline caches* [Hölzle et al. 1991]) static typing makes them much more efficient. Besides implementation techniques, substantial work has also been done to optimize object-oriented programs. For instance, the Vortex compiler is dedicated to the assessment of various optimization 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 many 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 implementations, besides PH and IC, they are generally not time-constant and 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, object-oriented implementation is not limited to method invocation, attribute access and subtype testing. A lot of little mechanisms are also implied—interested readers are referred to [Ducournau 2009] for a survey. Genericity is a major efficiency concern. The implementations of generics lie between two extremes [Odersky and Wadler 1997]. In *heterogeneous* implementations, e.g. C++ templates, each instance of the parametrized class is separately compiled. In *homogeneous* implementations, 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. A 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 situations, 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. .NET offers an intermediate implementation [Kennedy and Syme 2001] that should be markedly more efficient but still represents a research issue on JAVA platforms and for PRM.

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 for the implementation of any object-oriented language, as only common elementary mechanisms are tested. The conclusions apply in particular to production languages like C++, JAVA or EIFFEL.

When fixing a parameter, the tests provide an estimation of the difference in efficiency that must be expected when the other parameter varies. For instance, in a multiple inheritance setting, comparing G and D gives an estimation of the cost of dynamic loading. It also amounts to comparing EIFFEL and C++ that are closely related to their specific compilation scheme. Of course, C++ is only considered under the virtual implementation. In a dynamic loading setting, comparing S and D with attribute coloring is an estimation of the cost of JAVA-like multiple subtyping. In contrast, comparing attribute coloring and accessor simulation in D provides an estimation of the extra cost of full multiple inheritance in a dynamic loading setting. In practice, the results confirm that global compilation markedly improves the runtime efficiency, even when many optimizations are not considered like inlining. In this setting, the combination of coloring and BTD 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).

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 it a recommended technique 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, as its slight gain in method tables does not offset the slight time overhead and code length increase. Moreover, our tests show that an efficient underlying technique like PH must be preferred to caching.

In contrast, the conclusion concerning the mixed compilation scheme with link-time global optimization (O) is somewhat disappointing. A slight improvement was expected and the tests instead show a slight overhead on several processors. From the time standpoint, the link-time complication of these global optimizations might not be justified since 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 and space efficiency.

The tests were performed on a variety of processors, mainly with the same x86 architecture. Though most processors behave in a similar way, several exceptions lead us to conclude that language implementors should offer alternative implementations that might be customized on each specific computer and operating system. This would be especially useful and easy to carry out for virtual machines that rely on portable bytecode.

Attempt at Prescription. Of course, the findings of these experiments did not allow us to definitely decide on all processors and programs. The choice of an implementation will always depend on functional requirements such as dynamic loading. This article is not aimed at providing a prescription on how object-oriented languages should be implemented. However, starting from the initial three-fold tradeoff between modularity, efficiency and expressiveness, these first results can be formulated in a more prescriptive form as follows.

- If the point is efficiency, e.g. for a small embedded system, the solution is definitely global compilation (G), with a mixing of bounded BTD (restricted to BTD_1 if the processor is not equipped with branching prediction) and coloring. In this framework, the overhead of multiple inheritance is not significant.
- If the point is expressiveness, link-time coloring (S) certainly represents an interesting tradeoff between modularity and efficiency. It can be improved with link-time optimizations (O). If dynamic loading is not required, the hybrid compilation scheme that combines separate compilation of libraries and global compilation of programs (H) likely provides the best tradeoff between flexibility for rapid recompilations and efficiency of production runtimes. In this framework, the combination of coloring and BTD provides the most compact and efficient code.
- If the point is modularity, C++ is a proven solution in the framework of both multiple inheritance and dynamic loading, that could be improved with empty-subobject optimization. Moreover, compared to JAVA, the overhead of subobjects is in practice counterbalanced by template heterogeneous implementation and the fact that primitives types are not integrated in the object type system. This implementation presents, however, some scalability risks, as 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. Thus the conclusion may only hold for middle-size programs like the PRM compiler. In contrast, multiple subtyping represents an interesting tradeoff between expressiveness and efficiency. The efficiency of actual runtime systems mostly comes from JIT compilers but PH-`and` should be considered for the underlying interface implementation.

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

- 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 tested implementation techniques, especially coloring and perfect hashing, are not finely optimized from the space standpoint.
- The optimization of schemes O and G is not achieved, especially with regard to inlining for G, and dead code

elimination for O. Dead code elimination would be more difficult with O than with G, as usual linkers are not equipped for deleting code. The hybrid compilation scheme (H, Section 3.4) should also be tested.

- 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. Global link-time coloring (S) represents an efficient and simple fallback position.
- The subobject-based implementation must be achieved, completed with empty-subobject optimization (ESO). The alternative implementation with thunks [Ducournau 2009] should be considered too. Both are to be precisely compared to PH-`and` with both attribute coloring and accessor simulation. Anyway, the final implementation will never fully mimic C++, because of the PRM need for boxing and unboxing primitives types and its current homogeneous implementation of generics.
- 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 to take possible invariance into account, in a way similar to monomorphic calls.
- Polymorphic handling of primitive 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, as in .NET—this is a matter of further research for PRM.
- Apart from subobjects which might justify a fully conservative collector, a dedicated type accurate or at least semi-conservative garbage collection should reduce the overall time, while increasing the relative differences. This might reverse the conclusions of the comparison between subobjects and perfect hashing.
- Testing processors from other architectures is a 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, as 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++, C# and JAVA; with a clean integration of primitive types, like C#, 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 about 50% slower than the most efficient implementation with global compilation. Hence, 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.

D. 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 PLDI'93*, ACM SIGPLAN Notices, 28(6), pages 197–206, 1993.

D. 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 Conf. 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.

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

J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proc. ACM 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. Coloring, a versatile technique for implementing object-oriented languages. Rapport de Recherche 06-001, LIRMM, Université Montpellier 2, 2006.

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

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

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.

R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. Rapport de Recherche 08-017, LIRMM, Université Montpellier 2, 2008.

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.

S. 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.

R. Jones and R. Lins. *Garbage Collection*. Wiley, 1996.

A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proc. ACM PLDI'01*, pages 1–12. ACM Press, 2001.

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*, pages 17–32. Cépaduès, 2009.

H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer, 1993.

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.

K. Palacz and J. 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.

O. 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.

P. F. Sweeney and M. 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.

F. Tip and P. 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.