Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Towards a full multiple-inheritance virtual machine

Roland Ducournau[a] Floréal Morandat[ab]

a. LIRMM, Université Montpellier 2 and CNRS, France
http://www.lirmm.fr

b. S3L, Purdue University, IN, USA
http://purdue.edu

Abstract Late binding and subtyping create runtime overhead for object-oriented languages, especially in the context of both *multiple inheritance* and *dynamic loading*. Recent experiments show that this overhead is marked with static, non-adaptive compilers, which work under the *open-world assumption*. Therefore, dynamic, ie adaptive, compilation might present a solution to this efficiency issue. This paper presents the abstract architecture of a virtual machine and a dynamic compiler for unrestricted multiple-inheritance. This architecture involves an object representation that allows for shortcuts in the default implementations, coupled with compilation/recompilation protocols that maintain the most efficient implementations compatible with the current state of the program. The object representation proposed is based on *perfect class hashing*, which shortcuts to static calls or the single-subtyping implementation.

Moreover, this article proposes a new methodology, based on random simulation, for evaluating the runtime efficiency and recompilation cost of the proposed protocols. The resulting experiments show that the architecture proposed should provide the same runtime efficiency as Java and .Net, thus offsetting most of the multiple-inheritance overhead.

Keywords adaptive compiler, dynamic loading, late binding, method tables, multiple inheritance, open-world assumption, perfect hashing, random simulation, subtype test, virtual machine

1 Introduction

Multiple inheritance is generally considered as a cause of multiple difficulties from the standpoints of both semantics and runtime efficiency. Multiple subtyping, i.e. Java-like interfaces, was mostly proposed in response to these difficulties. Java popularised the idea [GJSB05], which has been widely adopted by recent languages, from the .Net family (e.g. C# [Mic01]) to the last Ada revision [TDB+06]. Moreover, some languages like Scala [OSV08] adopt a form of multiple inheritance, namely *mixins* (aka

Roland Ducournau, Floréal Morandat. *Towards a full multiple-inheritance virtual machine*. In Journal of Object Technology, vol. 11, no. 3, 2012, pages 6:1–29. doi:10.5381/jot.2012.11.3.a6

traits), which is midway between multiple-subtyping and full multiple inheritance. In this article, we do not address the semantic aspect of multiple inheritance, which is discussed in [DP11], and we focus on the efficiency aspect in a static-typing setting.

The main practical quality of the SCALA approach regarding multiple inheritance is that it makes the language compatible with multiple-subtyping platforms, which thus provide an easy and portable implementation. Indeed, although the overhead of multiple subtyping is not negligible—the so-called `invokeinterface` *considered harmful* [ACFG01]— it is generally agreed that the efficiency of JAVA and .NET systems comes from the fact that, in these languages, classes are in single inheritance and only interfaces present configurations similar to multiple inheritance. Their efficiency follows, too, from their dynamic, ie adaptive, compilation which allows for aggressive optimisations, at the expense of runtime recompilations.

In contrast, recent experiments show that the overhead of full multiple inheritance is marked in the context of dynamic loading and static, ie non-adaptive, compilation [DMP09]. Furthermore, we are not aware of a dynamic compilation framework dedicated to a language with static typing and full multiple inheritance. Therefore, the possibility of making full multiple inheritance as efficient as multiple subtyping is an open question, and positive answers likely involve dynamic compilation. This article is a first proposal towards an efficient runtime system dedicated to languages with multiple inheritance, static typing and dynamic loading. As an example, the proposed approach could be substituted to the current JAVA/.NET implementation of the SCALA language, and the specifications of this language could thus drop the class/trait distinction which seems to be inessential.

Object-oriented programming and, especially, multiple inheritance impact on runtime efficiency through three typical mechanisms, namely method invocation, attribute (aka field, slot, instance variable, ..) access, and subtype testing. All three mechanisms rely on the dynamic type of the receiver. Indeed, in a dynamic-loading setting which entails the *open world assumption* (OWA), multiple inheritance yields compile-time uncertainty about the position of accessed methods, attributes or supertypes. *Object representation* consists of the data structures that underlie the implementation of these mechanisms. Some of them are compatible with multiple inheritance and dynamic loading, but systematic experiments show how costly they are in a non-adaptive compiler, i.e. when the source code is compiled once for all [DMP09].

Dynamic compilation provides a general solution to this efficiency issue. It consists in an object representation that allows for shortcuts in the default implementations of these typical mechanisms, coupled with compilation/recompilation protocols that maintain the most efficient implementations compatible with the current state of the program, while keeping the recompilation cost reasonable.

This article presents the abstract architecture of a virtual machine designed for multiple inheritance which is expected to provide the same runtime efficiency and scalability as JAVA and .NET platforms. This architecture relies on three elements: (i) an object-representation based on perfect class hashing [Duc08, DM11], which precisely describes in an algorithmic way the decisions taken for deciding the implementation of a class; (ii) a compilation protocol that describes in the same manner, the way each mechanism-invocation site is compiled according to the current situation; and (iii) a recompilation protocol specifying the way methods or invocation sites are recompiled when the situation evolves. The article illustrates this abstract architecture with a few specific compilation/recompilation protocols. Besides the application to full multiple

inheritance, this articles aims at specifying simple compilation protocols that rely on static analysis instead of runtime profiling. Finally, we propose a general methodology for evaluating these protocols with random simulation. Indeed, the protocol behaviour is closely dependent on the specific class-loading and method-compilation orders, and an execution of a program would only provide a single order. Therefore, we propose instead a simulation that loads classes at random and provides a kind of approximate envelope of all possible executions, thus giving a better idea of the worst-case behaviour. The overall approach is tested on a benchmark that consists of the PRM compiler already used in empirical experiments at runtime [DMP09].

The structure of the article is as follows. Section 2 presents the point of object-oriented implementation, and describes the two techniques that underlie our proposal, namely single-subtyping implementation and perfect class hashing. The next section presents an abstract description of the proposed virtual-machine architecture. Section 4 specifies a first protocol, simple but not very efficient, which is an adaptation to dynamic loading of the double compilation proposed for attribute access in [Mye95]. This first protocol optimises only `self`-invocations, ie invocations whose receiver is typed by the current class. The random simulation approach is described. It is applied on a very simple abstraction of the PRM compiler, and its results presented and discussed. Section 5 presents a more accurate protocol family, which potentially optimises each invocation site in the most efficient way. A random simulation, now based on a more complete abstraction of the PRM code, is described. Its results prove that the proposed approach is promising and deserves more in-depth assessment. Finally, the last section puts the proposed approach into perspective, and lists known limitations and a few prospects.

2 Object-oriented implementation

With object-oriented programming, implementation is concerned with object representation, that is the object layout and the associated data structures and algorithms that support method invocation, attribute access and subtype testing.

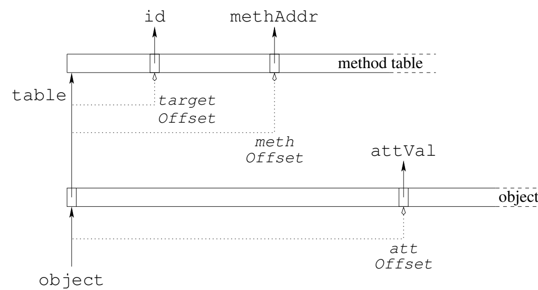
2.1 Single-subtyping implementation

In separate compilation of statically typed languages, late binding is generally implemented with method tables, which reduce method invocations to calls to pointers to functions through a small fixed number (usually 2) of extra indirections. Figure 1 depicts the overall data structure and the code sequence for each mechanisms. An object is laid out as an attribute table, with a `table` pointer at the method table, usually at offset 0 (`#tableOffset`). Each attribute has a fixed offset (`#attOffset`) in the object layout, while each method has a fixed offset (`#methOffset`) in the method table. With single inheritance and single subtyping, when classes are the only types, the class hierarchy is a tree and the tables implementing a class are straightforward extensions of those of its single direct superclass. Therefore, 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 (i.e. `#attOffset` and `#methOffset`) in their respective tables 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 efficiency of this implementation is due to both static typing and single inheritance. Otherwise,

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

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

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



Code sequences for the 3 basic mechanisms and the corresponding diagram of object layout and method table. The pseudo-code is borrowed from [Dri01]. Pointers and pointed values are in Roman type with solid lines, and offsets are italicised with dotted lines.

Figure 1 – Single-subtyping implementation.

the same kind of complication may occur when the same property name is at different places in unrelated classes.

The technique proposed by [Coh91] for subtype testing works under the OWA, too. 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 target class C if and only if the method table of x contains the class ID of C, at the position uniquely determined by C. Then, the test consists in comparing the content of the method table at the `#targetOffset` offset with the `#targetId`. Readers are referred to [Duc08] for implementation details that avoid bound checks and indirections.

The SST implementation is certainly the most efficient one under static, separate compilation. There are, however, almost no such SST languages. While this implementation is that of JAVA and .NET for class representation and class-typed invocations, it cannot take interfaces into account.

2.2 Perfect hashing

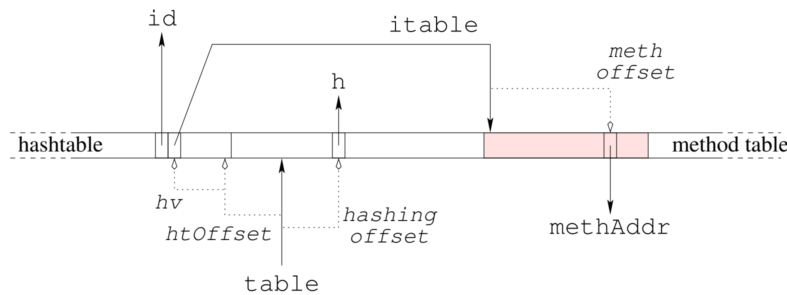
In [Duc08], we proposed a new technique based on perfect hashing for subtype testing in a multiple inheritance and dynamic loading setting. The problem can be formalised as follows. Let $(X, \preceq)$ be a partial order that represents a class hierarchy, namely X is a set of classes and $\preceq$ the specialisation 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. 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. If needed, still unloaded superclasses are recursively loaded. Then, the set $I_c = \{id_d \mid c \preceq d\}$ of the identifiers of all its superclasses is computed. Hence, $c \preceq d$ iff $id_d \in I_c$. This set I_c is immutable, and it can be hashed with a *perfect hashing function* h_c, i.e. a hashing function that is injective on I_c [CHM97]. The previous condition becomes $c \preceq d$ iff $ht_c[h_c(id_d)] = id_d$, whereby ht_c denotes the hashtable of c. All hashtables are immutable and the computation of ht_c depends only on I_c. The technique is thus incremental, hence compatible

```
// preamble
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #targetId, h, hv
sub table, hv, htable
```

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

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



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

Figure 2 – Perfect class hashing

with the OWA and dynamic loading. The perfect hashing functions h_c are such that $h_c(x) = hash(x, H_c)$, whereby the hashtable size H_c is defined as the least integer such that h_c is injective on I_c. Two *hash* functions were considered, namely *modulus* and bit-wise `and`[1]. A recent study led us to consider that the latter must be preferred [DM11].

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[2] by the interface. Of course, this easily generalises to method invocation in full multiple inheritance. Figure 2 recalls the precise implementation in this context. Like in SST, the object header points at its method table with the `table` pointer. The method table is bidirectional. Positive offsets involve the method table itself, organised as with single inheritance, whereby methods are grouped by introduction classes and these groups are arbitrarily ordered. Negative offsets consist of the hashtable, which contains, for each superclass d, a couple (id_d, m_d), where m_d is the address of the group of methods introduced by d. `#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 superclass ID, which must be compared to the target class ID (`#targetId`), and the address `itable` of the group of methods introduced by the superclass that introduces the considered method. The table contains, at the

[1] With `and`, the exact function maps x to $\texttt{and}(x, H_c - 1)$.
[2] The "introduction" term is crucial here. A class *introduces* a method when it defines a method with a *new* name (or signature) that is not already defined in any of its superclasses.

position `#methOffset` determined by the considered method in the method group, the address of the function that must be invoked.

The efficiency of PH is rather good. From the time standpoint, experiments in Prm show that the PH overhead is real but low, when it is used for method invocation and subtype testing [DMP09]. From the memory occupation standpoint, exhaustive random simulations on large-scale hierarchies show that joint computation of class IDs and PH parameters, called *perfect class numbering*, yields quasi-linear hashtables [DM11]. To our knowledge, PH is the only incremental, 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.

2.3 Accessor simulation (AS)

Originally, perfect class hashing was not intended to deal with attribute access, thus restricting its use to multiple subtyping. However, *accessor simulation* is a way of overcoming this restriction.

An accessor is a method that either reads or writes an attribute. 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* in [ZG03]. 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 offset used by Cohen's test (Fig. 3) [Mye95]. With PH, the attribute-group offset is associated with the class ID and method-group offset in the hashtable, yielding 3-fold table entries.

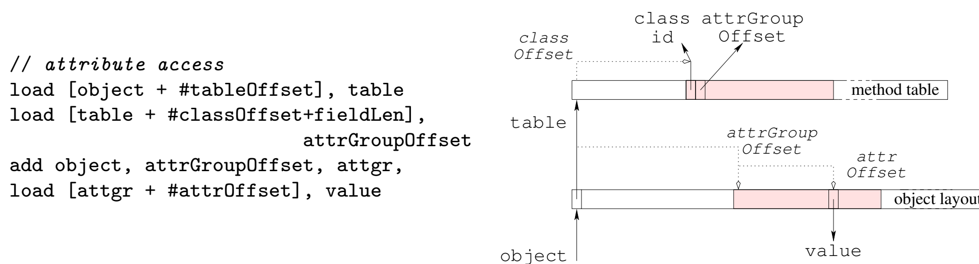
Accessor simulation is a generic approach to attribute access which works with any method invocation technique, and makes it work in a full multiple-inheritance setting if it works with multiple-subtyping. Only static typing is required, because of the merging of method addresses and attribute offsets in the same area, and since attributes must be partitioned by the classes that introduce them, hence introduced by a single class.

Among the various implementation techniques, some apply only to method invocation and subtype testing, e.g. perfect hashing. These techniques can thus be used for Java interface implementation. Accessor simulation is a way of applying them to full multiple inheritance. Whereas accessor simulation is a functional solution, it is not that efficient, and the Prm experiments [DMP09] showed that it yields marked overhead. However, this overhead is similar to that of C++-like subobjects [MD10].

3 Virtual machine specifications

In our abstract view, a virtual machine is a runtime system which performs the following tasks: (i) it loads code units that generally consist of classes that are not loaded yet; (ii) it computes the object representation for the newly loaded classes; (iii) it compiles or recompiles pieces of code (generally methods); and, finally, (iv) it runs the compiled code which can, cyclically, trigger class loadings and method recompilations. Of course, an actual virtual machine is markedly more complex, but we apply, here, Dijkstra's *separation of concerns* [Dij82].

```
// 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 Cohen's test, to be compared with Fig. 1. 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 offset (`#attrOffset`) that is now relative to this attribute group.

Figure 3 – Accessor simulation with Cohen's test

Furthermore, compilation is both lazy and adaptive. Laziness and adaptiveness are key features that distinguish modern runtime systems from the compilation setting that underly the implementation techniques presented in Section 2. Indeed, these techniques can be used with *static* compilation, ie without any need for recompilation. In contrast, the efficiency of Java and .Net systems rely on *dynamic* compilation, which can yield further recompilation. The code is generally loaded and compiled "just-in-time", and the compilation depends on the current state of the world. Therefore, each piece of code is compiled under a provisional *closed-world assumption* (CWA) which allows for efficient code sequences for the invocation of object-oriented mechanisms but can be invalidated by further class loadings. In contrast, we only consider object representations that do not need any runtime recomputation, hence which can be computed under the *open world assumption* (OWA). We thus attempt to take the best from the two worlds, ie static structures and dynamic code sequences.

Overall, the virtual machine specifications consist of two parts:

- an object representation that supports two kinds of alternative implementations for mechanism invocation: (i) a general background implementation is required to work in any situation and to present very good worst-case efficiency; (ii) one or more optimised implementations represent shortcuts with excellent efficiency, but they do not work everywhere and every time;
- a protocol for selecting the appropriate implementation and propagating possible recompilations, when the compiler must switch from an optimised implementation to a less optimised one.

3.1 Object representation.

The general idea is to use perfect class hashing as the underlying object representation, in such a way that all mechanisms could be invoked through PH, which is thus coupled with accessor simulation for attribute access. This base implementation allows for various shortcuts. The single-subtyping code can be used when the current situation satisfies the position invariant, i.e. when the considered property has the same position in all the subclasses of the receiver's static type. Moreover, monomorphic call sites could be implemented with static calls.

It is thus expected that PH will be used in very few cases, and that most invocation sites would use SST implementation, or even static calls. Indeed, all previous

experiments show that this goal should be reached to a large extent. Therefore, the issue is twofold: (i) how could optimisation improve the resulting efficiency? (ii) will the recompilation cost be acceptable?

Superclass ordering. Optimising the resulting efficiency is a matter of superclass ordering. Indeed, with PH, attributes and methods are grouped together in object layouts and method tables according to their introduction class. When a class is loaded, its attribute and method groups are determined according to its superclasses, and these groups must be ordered according to some algorithm. The resulting order is immutable, i.e. it will not be changed by further recompilations. Perfect class hashing is also computed and the complete method table can thus be allocated and filled (this is necessary only for concrete classes). The method-table structure, too, is immutable as only the entries corresponding to method addresses can be further changed.

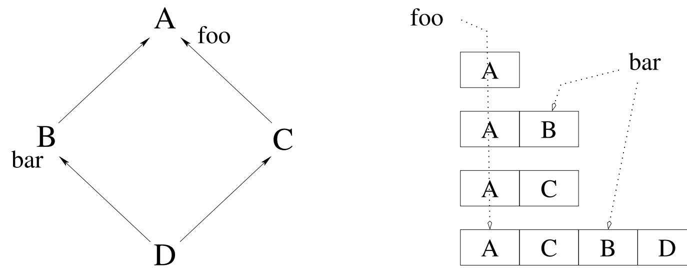
In contrast with these immutable structures, the order associated with the newly loaded class, say D, may assign to a superclass of D, say B, a position that differs from the single previous position of B. This will trigger some recompilations in B and other classes, when the SST code used for invocations involving B becomes unsound, hence forcing theses invocation sites to be compiled with the less efficient PH code. The efficiency of the approach thus depends on the superclass order.

It can be understood as a *linearization*, in the sense of *multiple-inheritance linearizations* [DP11]. Whereas multiple-inheritance linearizations are bottom-up, the superclass orders considered here are top-down; moreover, instead of being linear extensions of the inheritance relationship, they are ruled by the *prefix condition* as follows. Given a class c, with the set $sup(c)$ of its direct superclasses, then there must be some $c' \in sup(c)$ such that the superclass order of c' is a prefix of the superclass order of c, and c is said to satisfy the prefix condition w.r.t. c'. Therefore, loading c does not move c' and its superclasses, and they do not require any recompilation. In contrast, the superclasses of c that are not superclasses of c' will have multiple positions, and it might trigger some recompilations. In practice, the prefix selection follows some heuristics. For instance, it might be the class c' in $sup(c)$ that maximises the number of invocations that are at a constant position and should be optimised.

Let $sco(c)$ and $lin(c)$ denote, respectively, the superclass order and the aforementioned bottom-up linear extension associated with c. Let also $\oplus$ be the concatenation operation that removes duplicates and keeps the last occurrence of each. Then, the reverse superclass order of c, denoted $sco^r(c)$, is defined by $sco^r(c) = lin(c) \oplus sco^r(c')$. Because of this inversion, the prefix in the object representation is translated into a suffix in sco^r. It is easy to verify that $sco^r(c) = lin(c)$ in the single inheritance case, thus yielding the SST implementation.

There is no need for the superclass order to be the same for methods and attributes, since the hashtable is the only common point between the two kinds. These heuristics can also be improved by taking classes that introduce no attributes or methods into account.

Determining the position invariant. It might be too complex to accurately check the invariant condition, i.e. that the considered position is invariant in all subclasses of the receiver's static type, because it would require a data structure whose size is quadratic in the class number. There is, however, a simple, linear way to efficiently relax this condition. Each class keeps its single position in all of its subclasses, or a distinguished value (eg negative) when the invariant position is no longer satisfied. Now, an invocation site satisfies the position invariant if the invoked method (or



The picture depicts the smallest multiple-inheritance example, with a 4-class diamond and their associated tables which follow the corresponding superclass orders. Here, AC is a prefix of $ACBD$.

Figure 4 – Diamond example.

accessed attribute) is determined by a class with a single position.

When considering an invocation site like "`x:A ; x.foo()`", two possibly different classes can be considered for determining the position of `foo`, namely the receiver's static type (*rst*), i.e. `A`, and the property's introduction class (*pic*)[3], i.e. the *rst* or its superclass that introduces `foo`. The position invariant can be true with any of them, and false with the other[4].

Example. Consider for instance the famous multiple inheritance diamond of Figure 4. Suppose that the linearizations of these classes are, respectively, A, BA, CA and $DCBA$. Suppose also that C has been selected in $sup(D) = \{B, C\}$, so that D satisfies the prefix condition w.r.t. C. Then, $sco^r(D) = DCBA \oplus CA = DBCA$, and B has at least two different positions in B and D, since their superclass orders are, respectively, AB and $ACBD$. Finally suppose that A and D have a single position, eg all their other subclasses are in single inheritance, and that A and B introduce, respectively, `foo` and `bar`. Then, in the invocation site "`x:B ; x.foo()`", A is the *pic* with a single position, and B the *rst* with multiple positions. By contrast, in "`y:D ; y.bar()`", B is the *pic*, with multiple positions, and D the *rst* with a single position. Anyway, both sites can use the SST code since both `foo` and `bar` have a single position in these sites. In contrast, in the site "`z:B ; z.bar()`", *pic* and *rst* are B, which has multiple positions, hence `bar` has multiple positions, and the PH code must be used.

3.2 Compiling an invocation site

Without loss of generality, a variety of invocation sites can be distinguished from each other at load- or compile-time. The first two kinds are static, i.e. immutable and

[3] In static typing and multiple subtyping or inheritance, the *pic* is essential, and its uniqueness is the basis of the semantics proposed in [DP11]. However, not all languages conform with this uniqueness requirement. When a language accepts that a method has multiple *pic* (Java for instance), the method will have an entry in the method group of each *pic*, and these entries will be filled with the same method address.

[4] The proof of invariance is easy for the *pic* since a method/attribute has a fixed position in the method/attribute group of its *pic*. It is not so trivial for the *rst*, since it supposes that the *pic* has an invariant position in the implementation of the *rst*, but possibly not in other *pic* subclasses. The invariant is actually due to the superclass order defined above. Roughly speaking, if the *rst* has a fixed position, then it satisfies the prefix condition for all its subclasses, hence the *pic* will have a constant position in all the *rst* subclasses.

independent from the program execution, and the other are dynamic and mutable.

- A `self`-*invocation* is a method invocation or an attribute access whose receiver is the current receiver (called `self` in SMALLTALK, `this` in JAVA, C# and C++, or `Current` in EIFFEL). It can be generalised to the case where the *rst* is the including class (or a subtype of it, but the considered method or attribute must have been introduced by it). We consider only, here, the latter case, but the former could be considered for other optimisations like method *customisation* [CU89] (which makes `self`-invocations monomorphic).

- A *root-invocation* is a method invocation whose *pic* is the hierarchy root (i.e. `Any` in EIFFEL or PRM, or `Object` in JAVA or C#; of course, it cannot apply to C++). Indeed the root has always a single position, hence PH is never necessary for root-invocations. Moreover, this does not concern attributes, since the root has usually no attribute.

- Method invocation sites are often degenerate, in the sense that a single method can be invoked; the site is said to be *monomorphic* and can be compiled as a static call.[5]

- The site *rst* may be unknown, because it is not loaded yet; we call this case a `null`-*invocation*, since the receiver itself must be `null`. Hence, executing such a site must signal a `null`-exception. Of course, it cannot be a `self`-invocation, but it might be a root-invocation. If this is not a root-invocation, the invoked property is also unknown, and there is nothing to do at compile-time, although compiling it as a static call is a way to prepare the future.

- However, in a `null` root-invocation, the invoked method is known[6], since the root is always loaded first, and the call site can be definitively compiled as an SST invocation. It would reduce the recompilation number while, however, reducing the number of sites compiled as static calls.

- Finally, invocation sites that are neither monomorphic, nor root, nor `null`, can be compiled with SST or PH, according to the position invariance of the site.

A similar classification is done for attribute access, but it is simpler since, usually, the root does not introduce any attribute and monomorphism does not concern attributes. Thus, besides `self`-invocations, there remain only three cases—`null`, SST, and PH—which form a partition.

For subtype tests, there are three cases, too: (i) an equivalent of a monomorphic invocation is a test which succeeds (or fails) in all executions; the test always succeed when the target class is a superclass of all concrete subclasses of the `rst`; it always fails when the target class is not loaded yet (`null` case); (ii) SST implementation, i.e. Cohen's test, works when the position invariant holds for the target class; and (iii) PH is used in all other cases. However, as subtyping tests are negligible in our benchmark, we will no longer consider them.

[5] Here, we consider only call sites that are currently monomorphic but could become polymorphic in the future. Sites that are always monomorphic because the receiver's dynamic type is known at compile time, eg after intraprocedural control flow analysis, must be treated as a static call, statically, i.e. at load time.

[6] It might, however, be wrong with languages supporting static overloading, when the type of an actual parameter is a strict subtype of the corresponding formal parameter in the root. Indeed, the unknown *rst* might overload the root method with a more specific formal parameter type.

The overall data structure must record the invocation sites that may be further recompiled, i.e. all sites except those compiled with SST for root-invocations or PH for other sites. For instance, each class with a single position must maintain the list of already compiled methods that contain invocation sites whose *rst* or *pic* is this class. A similar data structure is required for monomorphic and `null` sites. These data structures are not that expensive, since the runtime system must maintain a model of the whole hierarchy anyway.

3.3 Compilation schedule

So far, we specified what to do, not when it can or must be done. Without loss of generality, a class must be loaded, and its representation computed, before it is instantiated, either directly via a `new`, or indirectly, when a proper subclass is instantiated. In a similar way, a method must be compiled or recompiled before it is invoked. More precise specifications are a matter of protocols which can mix both eagerness and laziness.

We do not specify, either, the way methods are compiled and recompiled. For instance, a method can be compiled as a whole, or piecewise. An invocation site can be inlined in the method code, or instead compiled as a stub function (aka a *thunk* or *trampoline*). It is worth noting that the invoked method is called by the thunk via a *tail call*, hence a thunk does not add an extra function call, but only an extra direct branch. On the contrary, a thunk used for an attribute access involves an extra function call, and is strictly equivalent to a monomorphic call to an accessor. Hence, a thunk used for method invocation entails almost no overhead, apart from possible cache misses, while a thunk used for attribute access or subtype testing involves the overhead of an actual function call. Besides this use, thunk or trampoline are also used for denoting the stub function that will trigger the compilation/recompilation of a method just before its next invocation. With this technique, a newly allocated method table is filled with a trampoline address, and the decision of recompiling a method just involves filling some method table entries with the trampoline address.

Another issue is left aside, though it is essential in practice, namely the recompilation of a method might be triggered while the method is active. It might occur even in synchronous executions. Furthermore, this recompilation might be mandatory in the sense that the current compiled version of the method would be erroneous. Consider for instance the fragment `x.foo().bar()`. The `foo` invocation may trigger the loading of a new class and return an instance of it, while the current compilation of the `bar` invocation has been made erroneous by this class loading. A common approach uses guarded implementations, like *polymorphic inline caches* [HCU91], which ensure that a background general implementation is available in the case where the guard is not satisfied. Our approach is different, since the protocols proposed hereafter always use the most efficient implementation in the current state of knowledge of the compiler. Therefore, this point must be carefully addressed, and techniques like *on-stack replacement* [FQ03, SKT07] might be considered, although we expect to avoid it, e.g. by replacing the `bar` invocation with a static call to a stub function which implements the right code sequence. This might also involve modifying the instruction cache.

4 Protocol for optimising `self`-invocations

A first protocol has been designed for optimising `self`-invocations, in order to make most of them as efficient as with SST. It tries to mimic the multiple-subtyping implementation in a static compilation setting, when class invocations are implemented with SST and interface-invocations with PH. However, with dynamic compilation and multiple inheritance, the invocations that are implemented with PH are determined by the recompilation protocol instead of being decided by the interface declaration.

In this protocol, when a method definition is compiled, all mechanism invocations are handled in the following ways:

1. subtype testing is implemented with PH;
2. all root-invocations use the SST implementation;
3. all `self`-invocations of methods or attributes are invoked through the SST implementation when the considered method or attribute group has still a single position;
4. PH is used for all other methods and attributes.

Monomorphic and `null`-invocations are not considered in a special way. Therefore, they are compiled as SST invocations. A particular method definition can use both SST for method invocation and PH for attribute access, or conversely.

Recompiling a class is required when the positions of its attribute or method groups have been changed in some subclass. It requires to change case-3 invocation sites into case-4. Only the methods that contain still optimised `self`-invocations are concerned, and a method can be recompiled at most twice, since the recompilation must be done only the first time the group is moved. However, the approach does not distinguish between `self`-invocations of methods/attributes introduced by the current class (which have actually been moved) and `self`-invocations of methods/attributes introduced in superclasses (which may have not been moved).

4.1 Protocols and algorithms

Recompilation protocol. Once a class d has been loaded, it maintains the following meta-information about its state:

- a_d and m_d represent the numbers of `self`-invocations of attributes and methods in the method definitions of d;
- ap_d and mp_d represent, respectively, the single position of the attribute or method group, or a distinguished value for multiple positions;
- A_d, M_d and AM_d represent sets of methods defined in d and candidates to possible recompilations; methods in A_d (resp. M_d) use the SST implementation for at least one attribute access (resp. method invocation), and methods in AM_d use SST for both.

When loading a class c, a direct superclass c' is first selected as a prefix. We take the class maximising the number $f(c') = \sum_{c' \preceq d} w_d * k_d$, whereby k_d is the number of considered invocations in the method definition of d, and $w_d = 1$ if d still has a single position, 0 otherwise. Each superclass d of c that is not superclass of c'

is then candidate to recompilation, and the methods in A_d (resp. M_d) and AM_d are recompiled if the attribute (resp. method) group has been moved. After recompilation, the A_d, M_d or AM_d sets of recompiled methods are removed. If recompilation concerns only method invocation (resp. attribute access), the AM_d set is then added to A_d (resp. M_d). The algorithm for deciding which methods must be recompiled is thus straightforward. As in all adaptive compilers, the recompilation itself can be eager or lazy.

Example. In the diamond example of Figure 4, class B has multiple positions after the definition of D, and all `self`-invocations in the methods defined in B must be recompiled.

4.2 Evaluation

The benchmark used in these tests is an abstract description of the class hierarchy of the PRM compiler used in the testbed presented in [DMP09].

Benchmark. This description follows the same principle as benchmarks commonly used in the object-oriented implementation community (eg [Dri01]), especially in our previous experiments [Duc08, DM11, Duc11]. It is, however, more detailed as it includes invocation counts. Each class is described by the following elements: (i) class name, (ii) superclass names, (iii) names of introduced attributes, (iv) names of defined methods followed by the number of invocations in the method code. Invocation numbers distinguish between method (M) and attribute (A) invocations. Method invocations are distinguished according to whether the method is introduced by the hierarchy root (R), the receiver is typed by the current class (S), or otherwise (O). The same is applied to attribute invocations, except they distinguish only between S and O cases, since there are generally no A-R example in any language. M-R represent cases that are always optimised, while A-S, M-S represent `self`-invocations that can be optimised and yield recompilations, and A-O and M-O are always unoptimised. The cardinality of the sets A, M and AM is thus computed for each class.

Compile-time statistics. Table 1 presents various statistics about (i) the number of attributes and methods, (ii) the number of invocation sites per invocation kind (A or M; R, S or O), and (iii) the total sizes of the A, M and AM sets, i.e. the number of method definitions according to the kinds of invocations they contain.

Random load-time statistics. Whereas these first statistics are static, i.e. they could be done at compile-time on the whole hierarchy (under the CWA), the next statistics are dynamic and they should be done at load-time. Therefore, class loading has been simulated, and class-loading orders are generated at random as in [DM11]. For each class-loading order, the number of recompiled classes, recompiled methods, variable-position methods and attributes, and variable-position invocation sites are computed, and Table 2 depicts the statistics on these numbers (minimum, average and maximum values).

Discussion. The most interesting numbers are the invocation numbers (Table 1(c)). As a consequence of a strict encapsulation discipline, almost all attribute accesses are `self`-invocations, hence concerned with the optimisation. In contrast, only 60% of method invocations are concerned with the optimisation. The random simulation

Table 1 – Self-invocation protocol — Compile-Time statistics

(a) Method numbers

introduced			defined			inherited		
total	avg	max	total	avg	max	total	avg	max
2880	5.0	107	4725	8.3	109	46204	80.8	237

(b) Attribute numbers

introduced			inherited		
total	avg	max	total	avg	max
629	1.1	29	2623	4.6	31

(c) Invocation numbers

A-S	A-O	A-*	M-R	M-S	M-O	M-*
5043	15	5058	19849	4182	15861	39892

(d) Number of methods according to their *self*-invocations

A	*AM*	*M*	none	total
1644	647	944	1490	4725

(a) Method numbers present total, per-class average and maximum, and distinguish between whether methods are introduced, defined or inherited in the class.
(b) The same applies to attributes, apart from the differences between introduction and definition.
(c) Invocation numbers retain the distinction between the different kinds made in the benchmark, i.e. root/self-invocations.
(d) The A, M and AM sets represent methods that contain self-invocations on attributes, methods or both.

Table 2 – Self-invocation protocol — Load-Time statistics

(a) Numbers of class and method recompilations

class rcp			meth rcp			rcp load		
63	66.5	68	360	363.1	389	31	33.5	36

(b) Variable-position `self`-invocations

variable A-S			total	variable M-S			total
121	121.1	140	5043	659	666.5	682	4182

Each random datum is depicted by its minimum, average and maximum values over all tested class-loading orders.
(a) The three columns depicts the number, respectively, of classes that define a method that must be recompiled, of methods that must be recompiled, and of classes whose loading triggers a recompilation. (b) The first part displays the number of `self`-invocations on attributes that cannot be optimised, and recalls the total number of `self`-invocations for attributes. The second part displays the same data for methods.

(Table 2(b)) shows that only a few percent of attribute and 16% of method self-invocations must finally be implemented with PH. Therefore the resulting program would be as efficient as JAVA or .NET systems for attribute access, as less than 3% of attribute access sites would not be optimised. In contrast, only 41% of method call sites would remain unoptimised, and this would looks like a JAVA program making an heavy use of invokeinterface.

The load-time recompilation cost is also interesting. At most 36 class loadings yield some recompilation on a total of 572 ('rcp load' column in Table 2(a)), and these recompilations concern at most 68 classes ('class rcp' column) and 389 methods ('meth rcp' column), on 3235 methods containing self-invocations, and a total of 4725 methods. Recompilation should thus concern less than 8% of the methods. This is an upper bound, since compilation would be lazy, and recompilation of a method might be triggered even before its first compilation.

5 More accurate recompilation-protocol family

In a more accurate compilation/recompilation protocol, each invocation site is compiled in the most efficient way according to the currently valid assumptions. Here, this protocol mimics global, static compilation with the proviso that the object representation depends on the class loading order, instead of being the result of global optimisation.

5.1 Protocol

Site-level protocol. This protocol distinguishes between all of the invocation cases listed in Section 3.2, apart from self-invocations. There remain five cases.

1. null-invocations are compiled in a special way discussed hereafter, and will require further recompilation when the invoked method will be known;
2. all monomorphic invocations are compiled as static calls, and further method overriding will need recompilations;
3. other root invocations, ie when they are polymorphic and non-null, are compiled with SST, without any further recompilation;
4. polymorphic non-root invocations are compiled with SST when the receiver's static type has a single position, and a recompilation will be needed when the condition is false; as mentioned above, the position invariant can be determined by the *rst* (rst case), the *pic* (pic case), or both (r+p case);
5. finally, polymorphic non-root invocations are compiled with PH when the *rst*, the *pic*, or both have multiple positions.

For attribute access, the partition between null, SST, and PH sites is done.

Null-invocations. The case of null-invocations deserves special consideration. First of all, they are not intended to be actually invoked, since the receiver of a null-invocation is always null and must be trapped by the implied null-check preceding the invocation. However, most null-invocations are provisional and destined to be transformed into an actual invocation. Therefore, instead of generating an empty

code sequence, it is preferable to generate the most probable sequence, with empty places that will be filled when a recompilation will be triggered. This should avoid a lot of full recompilations of the enclosing method. Thus, `null`-invocations will be compiled with an SST sequence for an attribute access, and a monomorphic call for a method invocation. When the recompilation is only triggered by the `null`-invocation, the empty place in the code sequence is filled, respectively, with the attribute position or the method address.

The case of root `null`-invocations is even special, since the SST implementation can avoid, here, any further recompilation, and might be preferred if the recompilation cost is too high. Therefore, root `null`-invocations can be managed in three different ways: (i) as ordinary `null`-invocations (`null` case); (ii) compiled with SST and never recompiled (`norcp` case); or (iii) compiled with SST and recompiled when a recompilation is triggered by another call site (`rcp` case).

Prefix selection. The selection of the direct superclass for the prefix condition should be optimised in the same way as in the `self`-invocation protocol, by attempting to minimise the exact number of PH invocations and/or the recompilation number. However, an exact evaluation of these criteria would be costly, and we used, in practice, the same selection as in the previous protocol.

Compilation schedule. Finally, the compilation protocol can be eager or lazy, and laziness should of course be preferred, in practice. However, as our simulation does not provide any control flow, our lazy protocol has a coarser grain, and a method is compiled as soon as an invocation site has been already compiled, or is in the compilation queue. Therefore, recompilation is always eager because the program abstraction used for simulations does not provide any control flow.

5.2 Simulation

This new protocol has been simulated on an extended abstraction of the benchmark used in the previous section, which is available on `http://www.lirmm.fr/~ducour/Benchmarks/benchmarks.html`. A method description now contains explicit descriptions of all invocation sites in the method body. The only difference between this abstraction and an actual program, is the program control-flow, which has been dropped, as a method body is just a set of invocation sites. As the program start-point (ie the main procedure) is available, static type analyses like *Rapid Type Analysis* (RTA) [BS96] are possible. In contrast, the lack of control flow makes interprocedural *Control-Flow Analysis* (CFA) impossible [Shi91].

Eager-protocol simulation. In the eager case, RTA is first applied to the benchmark program, statically. The general algorithm is as follows.

1. RTA produces a concrete-class set (*ccs*), and a live-method set (*lms*);
2. a class is taken at random in *ccs*, and its still unloaded superclasses are collected, hence leading to a set of unloaded classes to load (*ctl*), which is removed from *ccs*;
3. for each class in *ctl*, the implementation is computed; it yields a set of methods that must be recompiled because the *rst* of a `null`-invocation is now loaded, or because the *rst*/*pic* position of some sites has changed;

4. for each class *c* in *ctl*, the methods in *lms* and defined in *c* are collected, thus yielding the set of methods that must be compiled for the first time; this also yields another set of methods that must be recompiled because some monomorphic sites become polymorphic;

5. finally all the collected methods are compiled or recompiled, and the simulation loops at step 2.

Lazy-protocol simulation. In the lazy case, the simulation is closer to an actual execution. Now, the type analysis used is *Class Hierarchy Analysis* (CHA) [DGC95]. It proceeds as the simulation goes along, and yields the same final result as RTA in the eager protocol.

1. A set of still unloaded concrete classes (*ucs*) is maintained throughout the simulation; it is initialised with the class defining the program entry-point;

2. same as step 2 in the eager case, with *ucs* substituted for *ccs*;

3. same as step 3 in the eager case;

4. a set of methods to compile (*smc*) is now collected; it consists of methods that are defined in the classes in *ctl*, and are already invoked in previously compiled methods or in the current *smc* method-set; it also consists of methods previously defined but not compiled yet, that are invoked for the first time in the *smc* method-set;

5. in each method in *smc*, the instantiation sites (i.e. `new C`) are collected, and for each one, `C` is added to the *ucs* concrete-class set if it is not loaded yet;

6. finally all the collected methods are compiled or recompiled, and the simulation loops at step 2.

5.3 Evaluation

For each simulation determined by a single class-loading order taken at random according to one of these algorithms, the count of each invocation-site kind is computed. This was done thousands of times (actually 16150 for the data presented hereafter), for the complete protocol combinatorics, ie (i) lazy vs eager; (ii) `rst`, `pic` or `r+p` cases, which apply independently to methods and attributes; and (iii) with the root `null`-variants, ie `rcp`, `norcp` and `null` cases. From this large combinatorics, we present only the `r+p`/`r+p`/`null` triplet variant because it provides the best runtime efficiency and does not significantly increase the recompilation cost. The eager simulation is also presented for the sake of comparison.

Final implementation. Table 3 presents the statistics of all invocation sites at the end of each simulation, when all classes are loaded and all methods are compiled for ever. While the benchmark program is the same in all tests, the total number of invocation sites differ from the numbers in Table 1(c) because the optimisation of `self`-invocations does not involve type analysis and dead code elimination. In contrast, the protocols presented here involve type analyses, with algorithms which slightly differ between lazy and eager protocols. It is worth noting that the total numbers should be the same with lazy and eager simulations. The difference is an

artifact of our simulation, as the RTA algorithm used in the eager simulation is slightly more accurate than the lazy type analysis.

We first discuss the results of the lazy protocol. Table 3(a) concerns method invocations, and its first part root-invocations. On average, on a total of 17562 sites, 16843 are monomorphic and compiled with a static call, and the rest, ie 719, is compiled with SST. The next columns are dedicated to non-root invocations. On a total of 18928 sites, 2 are `null`-invocations, 15428 are monomorphic, 2663 are compiled with SST and 835 with PH. Regarding attribute access (Table 3(b)), the analysis is simpler, and on a total of 4429 sites, about 86 require PH. Moreover, some of these attribute invocations involve read/write accessors which could be optimised by redefining them in the subclasses that imply a different position for the attribute. The random effects are almost null, here. Indeed, the `r+p`/`r+p`/`null` variant is exact for root-invocations and monomorphic calls, and it happens that the effects on the SST/PH choice is negligible for method invocation, and very low for attribute access.

Overall, the resulting efficiency is very high, as there are less than 4321 sites, on a total of 40919 (i.e. 11%) that are not implemented in the most efficient way, i.e. SST for attributes and static calls for methods. Moreover, only the PH sites, which are less than 939, represent an actual overhead with respect to the best code that could be produced with SST under static compilation and global linking. This is negligible.

Recompilation costs. Table 4 presents the statistics of compilation and recompilation costs and numbers. Table 4(a) presents a few statistics on the benchmark program and the cost of a single compilation. These numbers are independent from the random simulation, but they actually depend on whether the compilation is lazy or eager, because of the slight aforementioned difference between the type analyses. In the lazy case, there are 3863 methods that must be compiled, and 870 among them do not require any recompilation because they do not include any invocation site (it can be determined statically). There are, respectively, 36490 and 4429 invocation sites for methods and attributes. Finally, we provided a rough estimation of the compilation cost of a method, as an affine function of its site number x, of the $ax + b$ form, where a and b are rather arbitrary coefficients. The absolute value is of course meaningless, and it is only used for the sake of comparison of the tested recompilation protocols.

Table 4(b) presents the statistics of the number of methods that require at least one full recompilation, along with the number of sites whose implementation is modified during this full method recompilation. Hence, this site number does not include `null`-invocation sites that are not recompiled but only filled. There are, on average, about 315 methods and 640 sites that are recompiled. Hence recompilation concerns less than 12% of methods, and 3% of invocation sites. Not surprisingly, the eager protocol involves many more recompilations.

Table 4(c) presents the statistics of the total number of recompilations, and the overall recompilation cost. The number of method recompilations is slightly higher, since different sites can trigger the recompilation of the same method, and it appears that, on average, a recompiled method is recompiled less than twice. In contrast, recompiled sites are generally recompiled once, since the single two-step transition, from monomorphic call to PH via SST, concerns very few sites. Finally, in the worst-case of lazy protocols, the overall recompilation cost is about 63% of the initial compilation cost. These numbers confirm that the numbers of recompiled methods and sites are not representative of the actual recompilation cost, since long methods are more often recompiled than short ones.

Table 3 – Accurate protocol — Final implementation

(a) Method invocations, final implementation

protocol	root			other		other SST		PH		total
	total	mono	SST	null	mono	min	max	min	max	
lazy	17562	16843	719	2	15428	2663	2664	834	835	36490
eager	16980	16262	718	3	14933	2591	2591	799	799	35306

(b) Attribute invocations, final implementation

protocol	null	SST			PH			PH in accessors		total
		min	avg	max	min	avg	max	min	max	
lazy	0	4325	4342.9	4343	86	86.1	104	8	10	4429
eager	0	4281	4283.0	4293	79	89.0	91	7	9	4372

Statistics on all invocation sites, when all classes are loaded and all live methods are compiled.
(a) Each line represents the statistics of the different kinds of call sites for a given protocol. These statistics include both constant numbers, which do not depend on the class-load ordering, and random numbers. The latter are depicted via minimum, average and maximum values. Average values are omitted when variations are low, and minimum and maximum values can be equal by chance.
(b) Similar statistics for attribute accesses.

Table 4 – Accurate protocol — Compilation and recompilation costs

(a) Compilation numbers

protocol	methods		sites			cost	
	all	norcp	meth	attr	all	all	norcp
lazy	3863	870	36490	4429	40919	145	2
eager	3801	865	35306	4372	39678	142	2

(b) Single-recompilation numbers

protocol	methods			sites			cost		
	min	avg	max	min	avg	max	min	avg	max
lazy	215	315.0	478	427	639.7	1136	20	30.0	48
eager	326	476.6	645	493	886.0	1331	23	37.7	51

(c) Full-recompilation numbers

protocol	methods			sites			cost		
	min	avg	max	min	avg	max	min	avg	max
lazy	277	409.9	741	427	643.5	1227	32	46.6	92
eager	422	633.3	874	507	908.5	1344	34	66.0	97

(a) The first column presents the numbers of methods that are compiled at least once, along with the number of methods that do not require any recompilation (`norcp`), because they do not contain any invocation site. The next column, depicts the number of invocation sites that are compiled at least once. Finally, an estimation of the static-compilation cost is presented; it is almost negligible for `norcp` methods.
(b) Statistics on the number of methods and sites that are recompiled at least once, and the corresponding recompilation cost if recompiled methods were recompiled only once.
(c) Statistics on the number of recompilation of methods and sites, and the corresponding recompilation cost.
(b) provides an estimate of the lower-bound of (c) in a lazier protocol.

First conclusions and comparisons. A general conclusion could be that these lazy protocols produce very efficient code, but at the expense of non-negligible recompilation cost when recompilation applies to each method as a whole.

In an actual compiler, recompilation would be lazy, and the comparison of lazy and eager compilation in Table 4(c) shows that laziness represents marked improvement. One might expect a similar gain with lazy recompilation. Table 4(b) displays the recompilation cost if recompiled methods were recompiled only once. This lower bound remains, however, about 33% of the initial compilation cost in the worst case. In a real compiler, however, all lazy compilations and recompilations would also occur later, just before the first invocation of the method considered. Hence, the recompilation cost would be lower, but we cannot estimate to what extent, since we just have an upper approximate of a lower bound.

Besides lazy/eager protocols, we must also compare the other parameters, namely the `rst`, `pic` or `r+p` criterion for using SST implementation, and the case of root `null`-invocations. In practice, `r+p` is slightly better than the `rst` variant, which is neatly better than the `pic` variant, and there is no real difference in terms of recompilation cost. Regarding root `null`-invocations, the code efficiency is clearly optimal with the `null` variant, since all monomorphic root-invocations are compiled as static calls, while the `rcp` and `norcp` variants do not recompile many of them. Moreover, the resulting increase in recompilation cost is almost negligible. This explains why we present only the `r+p`/`r+p`/`null` variant.

Partial recompilation and *thunks*. So far, the recompilation cost is marked because we consider only complete recompilation of methods. However, on average, when a method is recompiled, less than two sites are modified. Therefore, partial recompilation is certainly a worthwhile alternative. Furthermore, it might be a solution to the recompilation of a currently active method. A possible protocol could be as follows:

1. the initial compilation of each method is as usual with the `null` variant; each potentially recompilable site must record the address where its generated code sequence begins, but this is also needed for `null`-invocations with full method recompilation;

2. recompilation is managed at the site level and each kind of transition must be examined (the first two were already used above);

    - from `null` to monomorphic method invocation, the branch instruction is modified;
    - from `null` to SST attribute access, the SST sequence is completed with the right attribute offset;
    - from `null` or monomorphic method invocation to SST or PH, it involves generating a *thunk* which consists of the SST or PH sequence, and modifying the branch instruction;
    - from SST to PH, a thunk is generated for PH, and a branch instruction with a few `nop` instructions replaces the SST sequence;
    - from a SST thunk to PH, a new thunk is generated and the branch instruction modified.

Table 5 – Accurate protocol — Partial compilation and thunk number in the lazy protocol

(a) Thunk number

meth. thunk			attr. thunk			all thunk		
min	avg	max	min	avg	max	min	avg	max
404	626.4	1106	0	13.3	68	427	639.7	1136

(b) Method invocations, first implementation

root mono			root null			root SST		other null		other mono		other SST		PH	
min	avg	max	min	avg	max	min	max	min	max	min	max	min	max	min	max
3593	15237.4	16703	207	1684.6	13357	489	678	1892	6537	9771	14391	1385	2421	409	821

(c) Attribute invocations, first implementation

null			SST			PH		
min	avg	max	min	avg	max	min	avg	max
0	0.7	3	4325	4359.1	4417	12	69.2	104

(a) Statistics on thunk numbers.
(b-c) Statistics on all invocation sites after the initial compilation of each live method. The statistics are similar to those in Table 3, except that some numbers now depend on the class-load ordering.

3. of course complete and partial recompilation can be merged; partial recompilation is necessary when the recompiled method is active, and complete recompilation might be preferred when several sites must be recompiled.

In order to estimate the cost of such a protocol, Table 5(a) presents the statistics of invocation sites that call a thunk in the final implementation. Compared to the number of sites that might require a thunk according to Table 3, the number of thunks is very low. In the worst case, on 4303 invocation sites that are not optimised in the final implementation (ie PH sites plus SST method invocations), only 1136 sites (i.e. 26%) require a thunk. This is a measure of the way thunks degrade the runtime efficiency, and it can be considered as very low, since the unoptimised sites represent only 11% of all sites. Moreover, thunks can be shared when they address the same method/attribute; it might, however, degrade the efficiency of indirect-branching prediction.

Tables 5(b) and 5(c) present the number of sites of each kind at the first compilation of each live method. As they concern first compilation, these numbers do not depend on other protocol parameters than lazy/eager. In contrast with the final-implementation statistics (Table 3), they present marked deviation, especially for `null`-invocations which are not taken into account in the recompilation cost when it only involves filling the code sequence with the missing address/position. The recompilation cost is measured by the number of thunks, whose deviation is rather low, and `null`-invocations whose deviation is marked. However, thunks have a higher weight than `null`-invocations. Furthermore, the high number of `null`-invocations is likely an artifact of our random simulation, and they should be markedly lower in an actual execution.

Overall, both the runtime overhead caused by thunks and the recompilation cost would be quite negligible.

6 Related work, conclusions and prospects

In this article, we propose an abstract architecture of a virtual machine for full multiple-inheritance languages with dynamic loading, in a static-typing setting.

6.1 Contribution

The contribution of this proposal covers several related topics: (i) an object representation compatible with multiple inheritance and dynamic loading; (ii) the notion of load-time compilation/recompilation protocol as a first-class object of study; (iii) two specific recompilation protocols; (iv) a simulation methodology applied to these protocols; finally, (v) a generic program abstraction mostly independent of specific languages, which supports the simulation.

Object representation. The object representation proposed here is not new in itself, but only in its use in dynamic-loading settings. It is based on perfect class hashing [Duc08, DM11] and partially derived from the double compilation proposed by [Mye95], which allows the compiler to use the SST code sequences when the considered class has a constant position. However, Myers's double compilation was only concerned with attribute access and the choice was done at link-time under the CWA, whereas we extended his proposal to method invocation and applied it at load-time. To our knowledge, the only constant-time alternative to perfect class hashing is the subobject-based implementation of C++ (see for instance, [Lip96, Duc11]), but it is quite intricate, and not really more efficient than perfect hashing [MD10]. While this alternative represents a general background implementation compatible with dynamic loading, we don't see which shortcuts could optimise it. Another alternative involves *binary tree dispatch* (BTD) [ZCC97], which gives excellent results in a global compilation setting, especially when coupled with colouring for megamorphic sites [DMP09]. In a dynamic-loading setting, we expect with this technique a low improvement in efficiency, with a marked increase in recompilation costs. However, in view of our low simulation costs, an experiment would be worthwhile.

Finally, a common approach to method dispatch in a dynamic-loading setting is *polymorphic inline caches* [HCU91]. While it is certainly a solution with dynamic typing, we consider that static typing allows for more efficient techniques. Experiments on cache-based techniques [DMP09] show that they improve only on very inefficient underlying techniques. Moreover, inline caches are generally coupled with dynamic profiling, and recompilation never ends since the cache depends on the runtime behaviour. In contrast, our proposal does not require any dynamic profiling, and recompilation ends when all live classes are loaded. In view of the simulation presented here, we do not see any need for caches.

Compilation-protocol notion. All adaptive JIT compilers rely on some compilation/recompilation protocols (see for instance, [AFG+05]), but these protocols generally remain hidden and are not evaluated as such. For instance, in [IKY+00], the compilation techniques are described, but the protocol which decides which technique must be applied is not explicit. Moreover, the evaluation addresses only the runtime efficiency of the final code, not the recompilation cost. We consider that these protocols must be made explicit and their costs compared.

Specific compilation protocols. The protocols we propose concern only the three basic object-oriented mechanisms, namely method invocation, attribute access and subtype testing. Of course, adaptive compilers are not restricted to these features, and they address many other specific *hotspots* which are, however, out of the current focus of this work. The first protocol is close to Myers's aforementioned proposal. The only optimisation considered concerns `self`-invocations and the most optimised invocation sites are compiled with SST. The optimisation is poor, but the protocol is expected to be cheap.

The second protocol is markedly more ambitious, since it attempts to obtain the most optimised implementation for each site, without any a priori consideration regarding recompilation costs. It is not clear to what extent the proposed protocols are new in the details. For instance, the high frequency of monomorphic calls is well known, as is the thunk/trampoline technique. However, this technique seems to be used for lazy compilation, not for optimising method dispatch, and we did not find any mention of it in the literature, e.g. in the survey of adaptive compiler techniques [AFG+05]. In [IKY+00], a technique called *code-patching* is proposed for optimising monomorphic calls, by inlining the callee. Then, when a recompilation is needed, a precompiled code-patch replaces the inlined callee. Our thunk-based proposal thus represents a limited form of code-patching, where the patch concerns, in the worst-case, a predefined, short code sequence which is replaced with a jump followed by a `noop` sequence. In most of the cases, the patch concerns a single instruction. Conversely, code-patching could be used instead of thunks, by generating at the first compilation a `noop` sequence long enough for being fillable with the SST or PH sequence. It would, however, markedly increase the code size.

A similar thunk-based technique was proposed in a static compilation and global-linking setting [PD05], but real-size experiments were rather disappointing [DMP09]. Its use at load-time should not cause the same disappointment. Indeed, all invocations that are not monomorphic (i.e. 4219) must be implemented via a thunk at link-time, whereas only a very few (626) require a thunk at load-time. The overhead is even higher for attribute access, since all require a PH implementation in this approach.

Using static type analysis at runtime is not new in itself, and CHA is generally used for optimising monomorphic calls. We are, however, not aware of its use for optimising interface invocation, although a shortcut is possible when an interface-typed invocation site is polymorphic but the considered interface is directly implemented by a single class. More sophisticated type analyses are also possible, although their runtime cost might be markedly higher than CHA. For instance, in [QH03], the dynamic use of XTA [TP00] is proposed. However, in contrast with our approach, this proposal requires to extend method tables, so that method invocations require an extra indirection.

Random simulation. The assessment of such recompilation protocols implies to measure two kinds of parameters: (a) the runtime efficiency depends on the number of invocation sites for each kind of implementation; (b) the recompilation cost is a function of the number of methods and invocation sites that must be recompiled. These parameters must be computed during the execution of some benchmarks. However, developing a real runtime system for a language with multiple inheritance, in a dynamic-loading setting, would be exceptionally difficult. We did similar experiments in the context of static compilation [DMP09], and it involved a several-year work. In contrast, the simulation presented here has been developed in a few weeks, although on top of a simulation platform which is in progress for years.

Simulation is of course a common approach when physical experiments are impossible or too costly, and what is true for atomic bombs is also true for compilers, *mutatis mutandis*. However, simulation is not only a way to save time and pains, and randomness is an essential feature, too. First, it allows us to evaluate a family of executions instead of a single one, as it provides a kind of envelope of all possible executions of a given benchmark. Moreover, our previous study of perfect class hashing shows that several efficiency parameters depend on the precise class-loading order [DM11]. Our simulations confirm that both the efficiency and cost of these protocols closely depend on the class-loading order, i.e. on the program execution. Therefore, random simulation may well be the only way to evaluate their worst-case efficiency and resulting scalability. In contrast, runtime tests often represent only best-case assessments.

Generic program abstraction. Our benchmark consists of a real program, the compiler of the Prm language, which was also used in previous studies [DMP09]. However, our simulation does not directly work on the Prm code, but instead, on a language-independent abstraction which can be used for programs written in other languages. We are currently working on languages like Java, Scala and C# (along with Smalltalk in a different setting). The Prm benchmark and its syntax are available on `http://www.lirmm.fr/~ducour/Benchmarks/benchmarks.html`.

6.2 Comparison with runtime assessment.

In [DMP09], we presented empirical assessment of various implementation techniques and compilation schemes on the same Prm benchmark. In those experiments, the SST implementation can be obtained with attribute and method colouring, under separate compilation and global linking; it gives the MC-AC-S implementation (where MC, AC and S stand, respectively, for method colouring, attribute colouring and separate compilation) which serves as a reference in that article. Without recompilation, the object representation proposed here corresponds to PHand-AS-D, where PHand stands for perfect hashing with the bit-wise `and` hashing function, AS stands for accessor simulation and D means static compilation in dynamic loading. Notwithstanding the careful interpretation that those experiments require, especially regarding garbage collection and processors, and under the assumption that invocation sites are uniformly executed, one can extrapolate from those previous runtime experiments the expected runtime behaviour of the final code produced by our recompilation protocols.

On a recent processor such as Intel Core2 T7200, the execution time of the non-optimised PHand-AS-D version would be about 130% of the MC-AC-S reference time, which is about 30s. This is roughly the price to pay for multiple inheritance in a static compilation and dynamic loading setting.

The version resulting from the first recompilation protocol is midway between SST and PH for method invocation, and very close to SST for attribute access. Therefore, one can estimate its execution time as the mean of the MC-AC-S and PHand-AC-D measures, and the latter is about 105% of the reference time. The optimising factor of this protocol is thus about 0.8 (i.e. 102.5/130).

In contrast, the optimised version resulting from the second recompilation protocol is close to the MC-AC-BTD_0-G variant, where MC and AC stand respectively for method and attribute colouring, BTD_0 represents compilation of monomorphic calls as static calls, and G means global static compilation. On the same processor, the

optimised version would be about 90% of the reference time. Thus, the optimisation factor of this recompilation protocol would be about 0.7 (i.e. 90/130).

6.3 The multiple-inheritance benchmark issue.

Of course a single benchmark is not enough, and this is the main limitation of this work. This was already a criticism of our aforementioned study, but it was inherent to the fact that we were developing a new language, and its compiler was the only significant program written in this language [DMP09]. It is, however, arguable that this benchmark is significant, fully object-oriented, and representative of object-oriented programming. Indeed, the program statistics (see [DMP09]) present multiple similarities with statistics reported in the literature for other programs and other languages, regarding various parameters such as the proportion of monomorphic calls or the average number of invocation sites per method.

However, the question of multiple-inheritance benchmarks is, in itself, an open issue. Firstly, there is no object-oriented standard, especially in multiple inheritance. Interested readers are referred to [DP11] for an in-depth discussion of this point. Besides PRM, we could consider SCALA, C++ and EIFFEL programs. Each of them has, however, a specific point of view regarding multiple inheritance. Moreover, while SCALA and EIFFEL are pure object-oriented languages, as PRM or JAVA, this is not the case with C++, because it provides the ability to program at the C level, and because of the `virtual` keyword which allows programmers to shortcut late binding by hands. To a lesser extent, this reserve applies to C#, too. Accordingly, the statistics on monomorphic invocations presented in [SD12] markedly differ from those presented here.

Using a language-independent program abstraction will allow us to overcome this single benchmark issue, but extracting a complete abstraction from programs written in different languages remains a heavy task. Contributors are welcome.

Again, a last argument is random simulation. It explores many more possibilities than a single execution and it makes each benchmark far more representative than a single execution.

6.4 Related work

As already mentioned, the optimisations that are commonly undertaken by adaptive compilers are not restricted to object-representation shortcuts. Two of them deserve, here, some discussion, namely code specialisation and method inlining. Code specialisation involves duplicating a piece of code in order to optimise it in different contexts. It can be done at the method level, e.g. *customisation* [CU89], or at a finer grain, e.g. at the loop level as in [IKY⁺00]. When viewed as a source-to-source transformation, code specialisation is compatible with our approach, and it might improve the shortcut frequency; for instance, systematic method customisation makes `self` monomorphic.

Function inlining is one of the most common optimisation of programs, which cannot be used without restriction with object-oriented programming because of late binding. Therefore, it is generally used only for monomorphic calls. With dynamic recompilations, inlining raises the same issues as object-representation shortcuts. Actually, shortcuts represent inlined code sequences, and the only difference is that recompilation of an inlined method also affects the inlining methods. Anyway, the

main issue involves recompiling a method which is currently active. The aforementioned code-patching approach has been proposed in this context. Besides, the notion of *preexistence* has been proposed for deciding whether the current version can safely continue, or not [DA99]. It could be used in our framework, as well.

Both code-specialisation and method-inlining optimisations may, however, result in marked increase in code size, hence in compilation costs. Anyway, it would be easy to evaluate their contribution in our framework, at least when the optimisation is decided statically, i.e. without runtime profiling. Indeed, a distinctive feature of our recompilation protocols is that they do not depend on dynamic considerations that would require profiling. We use only static analyses, although they are performed dynamically. This is partly due to our simulation framework which cannot assess protocols based on runtime profiling, and partly due to our conviction that purely static protocols will be more efficient. This conviction, however, requires a different testbed to be verified.

6.5 Conclusions and prospects

Two protocol families have been specified and tested. The first one was a trial run, and it optimises only `self`-invocations. It shows that the resulting efficiency should be close to that of multiple-subtyping languages in a static compilation setting, from both run-time and load-time standpoints, i.e. with SST for classes and PH for interfaces. However, while it gives good results on its target, it lets all other invocations be implemented with PH, and they are too many. Therefore, it remains far less efficient than separate static compilation coupled with link-time colouring [DMP09]. Furthermore it does not consider monomorphic calls. Overall, this first protocol does not deserve to be further considered, at least for a pure object-oriented language where all function invocations rely on late binding. In contrast, languages like C++ and C# allow the programmer to specify (with the `virtual` keyword) whether a method must be invoked statically, or via late binding. Therefore, in these languages, monomorphic calls might be rare and this protocol interesting.

The results of the second protocol family are markedly more promising for pure object-oriented languages. First, most invocation sites can be implemented in the most efficient way, which specifically depends on each site. As a consequence, a majority of method invocations are treated as static calls. The recompilation cost is not negligible with full-method recompilation, but partial recompilation and thunks provide high efficiency at a negligible cost. Overall, this protocol seems to be an excellent tradeoff, since the resulting code and efficiency are very close to those of global compilation, and the recompilation cost is negligible. Of course, these conclusions hold for the tested benchmark and there remains to check that this protocol gives similar results on a variety of benchmarks representative of different programming styles.

The prospects of this work are manifold. (i) The proposed protocols could be extended to many other optimisations that are compatible with our simulation framework and commonly carried out by adaptive compilers. For instance, method inlining and method customisation are interesting candidates. Simulation would be a marked improvement in the assessment of the benefits and costs of these optimisations. In contrast, many other optimisations like loop optimisations could not be simulated in our framework. (ii) We have to achieve the adaptation of our program abstraction to other languages, starting with Scala, Java, Eiffel and C#. It will provide us with a lot of significant benchmarks, and a stronger assessment of the proposed protocols. (iii) Our main goal is to design and develop a runtime system for a language

with multiple inheritance and dynamic loading, that would implement such recompilation protocols. It should be an opportunity to complement our abstract assessment with empirical time measurement of actual program executions. Like the SCALA language, these new language and runtime systems should provide some compatibility with existing bytecodes, eg JVM or .NET, in order to reuse existing libraries. (iv) The recompilation protocols proposed here could be also of interest for existing JAVA and .NET runtime systems. Indeed, the question of monomorphic calls is universal in object-oriented programming, and interface-typed invocation site need shortcuts.

Finally, we are currently working on languages like SMALLTALK, that are in single inheritance and dynamic typing, for which we designed an object representation based on a novel usage of perfect hashing, with a recompilation protocol similar to the present one, apart from the lack of static types.

References

[ACFG01] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proc. OOPSLA'01*, SIGPLAN Not. 36(10), pages 108–124. ACM, 2001. doi:10.1145/504311.504291.

[AFG+05] 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. doi:10.1109/JPROC.2004.840305.

[BS96] D.F. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*, SIGPLAN Not. 31(10), pages 324–341. ACM, 1996. doi:10.1145/236338.236371.

[CHM97] Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theor. Comput. Sci.*, 182(1-2):1–143, 1997. doi:10.1016/S0304-3975(96)00146-6.

[Coh91] N. H. Cohen. Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4):626–629, 1991. doi:10.1145/59287.59293.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *Proc. OOPSLA'89*, SIGPLAN Not. 24(10), pages 146–160. ACM, 1989. doi:10.1145/74818.74831.

[DA99] D. Detlefs and O. Agesen. Inlining of virtual methods. In R. Guerraoui, editor, *Proc. ECOOP'99*, LNCS 1628, pages 258–277. Springer, 1999. doi:10.1007/3-540-48743-3_12.

[DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proc. ECOOP'95*, LNCS 952, pages 77–101. Springer, 1995. doi:10.1007/3-540-49538-X_5.

[Dij82] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, page 60–66. Springer-Verlag, 1982.

[DM11] R. Ducournau and F. Morandat. Perfect class hashing and numbering for object-oriented implementation. *Softw. Pract. Exper.*, 41(6):661–694, 2011. doi:10.1002/spe.1024.

[DMP09] R. Ducournau, F. Morandat, and J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In Gary T. Leavens, editor, *Proc. OOPSLA'09*, SIGPLAN Not. 44(10), pages 41–60. ACM, 2009. doi:10.1145/1639949.1640093.

[DP11] R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. *Science of Computer Programming*, 76(7):555–586, 2011. doi:10.1016/j.scico.2010.10.006.

[Dri01] K. Driesen. *Efficient Polymorphic Calls.* Kluwer Academic Publisher, 2001.

[Duc08] R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6):1–56, 2008. doi:10.1145/1391956.1391960.

[Duc11] R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comp. Surv.*, 43(4), 2011. doi:10.1145/1922649.1922655.

[FQ03] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proc. CGO'03*, pages 241–252. IEEE Computer Society, 2003. doi:10.1109/CGO.2003.1191549.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The JAVA Language Specification.* Addison-Wesley, third edition, 2005.

[HCU91] 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. doi:10.1007/BFb0057013.

[IKY+00] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proc. ACM OOPSLA '00*, pages 294–310, 2000. doi:10.1145/353171.353191.

[Lip96] S. B. Lippman. *Inside the C++ Object Model.* Addison-Wesley, New York, 1996.

[MD10] F. Morandat and R. Ducournau. Empirical assessment of C++-like implementations for multiple inheritance. In *Proc. ICOOOLPS Workshop*, pages 7–11. ACM, 2010. doi:10.1145/1639949.1640093.

[Mic01] Microsoft. C# Language specifications, v0.28. Technical report, Microsoft Corporation, 2001.

[Mye95] A. Myers. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*, SIGPLAN Not. 30(10), pages 124–139. ACM, 1995. doi:10.1145/217839.217849.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, A comprehensive step-by-step guide.* Artima, 2008.

[PD05] 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. doi:10.1145/1108792.1108799.

[QH03] F. Qian and L. Hendren. Towards dynamic interprocedural analysis in JVMs. Sable Technical Report 2003–5, McGill University, 2003.

[SD12] O. Sallenave and R. Ducournau. Efficient compilation of .NET programs for embedded systems. *Journal of Object Technology*, 11:27, 2012. `doi:10.5381/jot.2012.11.1.a2`.

[Shi91] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[SKT07] E. Steiner, A. Krall, and C. Thalinger. Adaptive inlining and on-stack replacement in the Cacao virtual machine. In *Proc. PPPJ '07*, pages 221–226. ACM, 2007. `doi:10.1145/1294325.1294356`.

[TDB+06] 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.

[TP00] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. OOPSLA '00*, pages 281–293. ACM, 2000. `doi:10.1145/353171.353190`.

[ZCC97] O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proc. OOPSLA'97*, SIGPLAN Not. 32(10), pages 125–141. ACM, 1997. `doi:10.1145/263700.263728`.

[ZG03] 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. `doi:10.1007/978-3-540-45070-2_15`.

About the authors

Roland Ducournau is Professor of Computer Science at the University of Montpellier. In the late 80s, while with Sema Group, he designed and developed the YAFOOL language, based on frames and prototypes and dedicated to knowledge based systems. His research topic focuses on class specialization and inheritance, especially multiple inheritance. His recent work is dedicated to the design and assessment of scalable constant-time techniques for implementing object-oriented languages. Contact him at `roland.ducournau@lirmm.fr`, or visit `http://www.lirmm.fr/~ducour`.

Floréal Morandat defended his PhD thesis in Montpellier in 2011. His thesis presents a systematic assesment of various implementation techniques and compilation schemes for object-oriented programming, based on a testbed formed by the PRM compiler. He is currently post-doctorant at Purdue University. Contact him at `fmoranda@purdue.edu`, or visit `http://www.lirmm.fr/~morandat`.