Link-Time Static Analysis for Efficient Separate Compilation of Object-Oriented Languages

Jean Privat
privat@lirmm.fr

Roland Ducournau
ducour@lirmm.fr

LIRMM
CNRS — Université Montpellier II
161 rue Ada
34392 Montpellier cedex 5, France

ABSTRACT

Compilers used in industry are mainly based on a separate compilation framework. However, the knowledge of the whole program improves efficiency of object-oriented language compilers, therefore more efficient implementation techniques are based on a global compilation framework.

In this paper, we propose a compromise by including three global compilation techniques (type analysis, coloring and binary tree dispatching) in a separate compilation framework. Files are independently compiled into standard binary files with unresolved symbols. The program is build by linking object files: files are gathered and analyzed, some link code is generated then symbols are resolved.

1. INTRODUCTION

According to software engineering, programmers must write modular software. Object-oriented programming has become a major trend because it fulfills this need: heavy use of *inheritance* and *late binding*[1] is likely to make code more extensible and reusable.

According to software engineering, programmers also need to produce software in a modular way. Typically, we can identify three advantages: (i) a software component (e.g. a library) should be distributable in a compiled form; (ii) a small modification in the source code should not require a recompilation of the whole program; (iii) a single compilation of a software component should be enough even if it is shared by many programs. In order to have these advantages, FORTRAN has introduced separate compilation: source files are compiled independently of future uses, then linked to produce an executable program.

The problem is that the knowledge of the whole program allows more efficient implementation techniques. Therefore previous works use these techniques in a global compilation framework, thus incompatible with modular production of softwares. Global techniques allow efficient implementation of the three main object-oriented mechanisms: late binding, read and write access to attributes, dynamic type checking.

In this paper, we present a genuine separate compilation framework that includes some global optimization techniques. The framework described here can be used for any statically typed class-based languages [17] like C++ [21], JAVA [12] and EIFFEL [18], but, it is not applicable to SELF [23] or SMALLTALK [11].

The remainder of this paper is organized as follows. Section 2 presents the global optimization techniques we consider. Section 3 introduces our separate compilation framework. Results and benchmarks they where obtained from are presented in section 4. We conclude in section 5.

2. GLOBAL TECHNIQUES

The knowledge of the whole program source code permits a precise analysis of the behavior of each component and an analysis of the class hierarchy structure. Each of those allows important optimizations and may be used in any global compiler.

2.1 Type Analysis

Late binding is considered as a bottleneck in object-oriented programs. Statistics show that most method calls are actually monomorphic calls. Type analysis[2] can detect these monomorphic calls and reduces the polymorphism of others.

A type analysis approximates three sets: the set of the classes that have instances (live classes), the set of the *concrete type* of each expression (the concrete type is the set of dynamic types) and the set of called methods for each call site. These three sets are mutually dependent: called meth-

[1]Instead of applying a function to arguments, a *message* is sent to an object, the *receiver*. The program behavior, i.e. the code which is executed, depends on the value of receiver. From an implementation point of view, it follows that the *static* function call of procedural language must be replaced by something more dynamic since the control flow jumps to an address extracted from the receiver value.

[2]Type analysis should not be confused with the ML type inference.

PASTE '05 Lisbon, Portugal

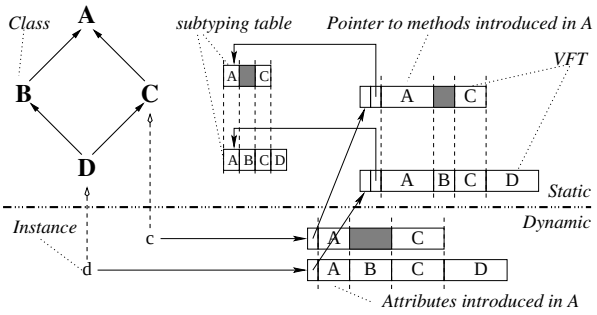

Figure 1: Implementation of Classes and Instances with Coloring

ods depend on the concrete type of the receiver, concrete types depend on the instantiated classes, and instantiated classes depend on the called methods. This interdependence explains the difficulty of the problem [10], and the variety of solutions [13]. There are many kinds of type analysis, even simple ones give good result and can detect many monomorphic calls [2].

2.2 Coloring

Multiple inheritance is problematic with the standard virtual function table (VFT) implementation. C++ resolves it by the use of subobjects and an important overhead [16, 8]: (i) in the worst case, the number of method tables is quadratic (instead of linear) and the size is cubic (instead of quadratic); (ii) with subobjects, in the dynamic memory allocated for an instance, the number of attributes can be less than the number of pointers to VFT; (iii) pointers to an instance are dependent of the static type of the pointers.

Coloring avoids the overhead of multiple inheritance. It can be applied to attributes, to methods and to classes for subtyping check [6, 20, 5, 24, 8]. The implementation of classes and instances includes two parts (Figure 1): in static memory, an area for each class with the address of each method (in the VFT) and the superclass information (in the subtyping table); in dynamic memory, an area for each instance with attributes and a pointer to its class. Figure 1 differentiates VFT and subtyping table but in the rest of the paper, both are merged.

The technique consists in assigning a unique identifier to each class and a *color* (an index) to each class, method and attribute. Colors are assigned in such a way that:

Invariant 1. *A pointer to an instance does not depend on the static type of the pointer.* Thus, polymorphic assignments and parameter passing are costless (as opposed to C++ pointer adjustments).

Invariant 2. *The color of an attribute (respectively a method) is invariant by inheritance and redefinition.* Thus, the index of the attributes (respectively methods) does not depend on the static type of the receiver.

Invariant 3. *Two classes with the same color do not have a common subclass.* The subtyping table of a class contains the identifier of each super-class at the index of this super-class.

For implantation example, assuming a polymorphic call site `x.f` and assuming the color of the `f` method is Δ_f, the generated code in assembler language looks like:

```
mov [x + #tableOffset] → table
mov [table + Δf] → method
call method
```

The same technique can be used for attribute access and type check. See section 3.1.3 for code sample.

Finding a coloring that respects the three invariants requires the knowledge of the whole class hierarchy. Minimizing the size of the table (i.e. minimizing the number of gaps, in gray in Figure 1) is an NP-hard problem similar to the minimum graph coloring problem. Happily, class hierarchies seem to be simple cases of this problem and many efficient heuristics are proposed in [20, 22].

2.3 Binary Tree Dispatch

SmartEiffel [25] introduces an implementation technique for object-oriented languages called *binary tree dispatch* (BTD). It is a systematization of some techniques known as *polymorphic inline cache* and *type prediction* [14]. BTD has good results because VFT does not schedule well on modern processors since the unpredictable and indirect branches break their pipelines [7].

BTD requires a global type analysis in order to reduce the number of expected types of each call site. Once the analysis is performed, the knowledge of concrete types permits to implement polymorphism with an efficient select tree. The select tree enumerates types of the concrete type and provides a static resolution for each possible case. This technique does not need a memory implementation for classes and the dynamic memory area for an instance contains the class identifier (an integer) and the attributes.

For example, let's consider a polymorphic call site `x.f` where the concrete type of x is $\{A, B, C, D\}$. Assuming the class identifiers of these classes are, respectively, 19, 12, 27 and 15, assuming their f method implementations are, respectively, f_A, f_B, f_C and f_D and assuming the identifier of the dynamic class of `x` is id_x, the generated code looks like:

```
if id_x ≤ 15 then
       if id_x ≤ 12 then f_B
       else f_D
       end
else
       if id_x ≤ 19 then f_A
       else f_C
       end
end
```

Obviously, the same technique can be used for attribute access and type check.

3. SEPARATE COMPILATION

Separate compilation frameworks are divided into two phases: a local one (compiling) and a global one (linking).

The *local phase* compiles a single software component (without loss of generality, we consider the compilation units to be classes) independently of the other components. We denote *binary components* the results of this phase[3]. *Binary*

[3]Traditionally, the results of separate compilation are called *object files*. Because this paper is about object-oriented languages, we chose to not use the traditional name to avoid conflicts.

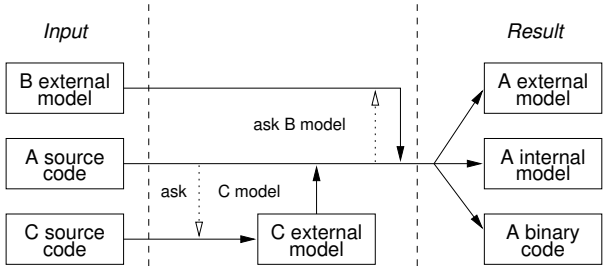

Figure 2: Local Phase

components are written in the target language of the whole compilation process (eg. machine language) but they are not functional because some missing information is replaced by symbols. The *binary components* also contain metadata: debug information, symbol table, etc.

The *global phase* gathers *binary components* of the whole program, collects some metadata, resolves symbols and substitutes them. The result of this phase is a functional *executable* that is the compiled version of the whole program.

The compilation framework proposed in this paper conforms to this general approach.

Other works, [9] and [3], propose a separate compilation framework with global optimization respectively for Modula-3 and for functional languages. In both cases, the main difference with our approach is that their local phases generate code in an intermediate language. On linking, global optimizations are performed on the whole program then a genuine global compilation translates this intermediate language into the final language.

3.1 Local Phase

The local phase takes as its input the source code of a class, and produces as its results the *binary code* and two metadata types: the *external model* and the *internal model* (Figure 2). These three parts can be included in a same file or in distinct files but the *external model* should be separately available.

3.1.1 External Model

The external model of a class describes its interface: superclasses and (re-)definitions of methods and attributes. The local phase compiles classes independently of the final programs but the compilation of a class needs the interface of its related classes: superclasses and used classes. Thus, the external model of these classes must be available or be generated from the source file. In the latter case, a recursive generation may be performed.

3.1.2 Binary Code

The binary code contains symbols. As in standard separate compilation, symbols are used for addresses of functions and global variables. In our proposition, we also introduce other symbols: resolution addresses of late binding sites, attribute colors, class colors, class identifiers, instance sizes, and class table addresses.

The local phase assigns a unique symbol to each late binding site. A site `x.f(a)` associated with a symbol `f12` is compiled with a static direct call:

```
push x
push a
call f12
```

Thanks to coloring, attribute accesses are compiled with a direct access in instance layout. For example, we assume that the symbol of the color of the attribute a is Δ_a. Reading the attribute a of the object x in the register val is compiled as:

`mov [x +` Δ_a`]` $\rightarrow$ val

Thanks to coloring, type checks are compiled with a check in the class table of the instance [5, 24]. For example, we assume that Δ_C and ID_C are respectively the symbols for the color and the identifier of the target class C. Checking that x is an instance of C or an instance of a subclass of C is compiled as:

```
mov [x + #tableOffset] → table
mov [table + ΔC] → class
cmp class, IDC
jne false
       check success
jmp end
false:
       check fail
end:
```

The creation of an instance needs a memory allocation in the heap and an assignment of the class table pointer at `#tableOffset`. Thus, two symbols are required, one for the size of the allocated memory, the other for the class table pointer.

3.1.3 Internal Model

The internal model of a class describes the behavior of its methods. There are two kinds of internal models according to the global type analysis performed at link time: with or without flow analysis.

In both cases, the internal model of a method gathers class instantiations, late binding sites, attribute accesses and type checks. The internal model also associates them with symbols used in the binary code and with static types used in the source code.

Using a type flow analysis, the internal model of a method also contains a graph which represents the circulation of the types between the *entries* and the *exits* of the method. An *entry* is the receiver, a parameter, the reading of an attribute, or the result of a method call. An *exit* is the result of the method, the writing of an attribute, or the arguments of a method call. The vertices of the graph are the entries, the exits, the class instantiations, and some intermediate nodes that represent expressions and local variables. The edges of the graph are the inclusion constraints for the concrete types associated with each vertex.

An intraprocedural analysis makes it possible to build these internal diagrams by minimizing their size. The analysis may be limited to polymorphic types, since the concrete type of monomorphic types, in particular primitive types, is statically known. This type circulation graph corresponds to the *template* introduced by [1] even if intraprocedural and interprocedural analysis were not separated in time.

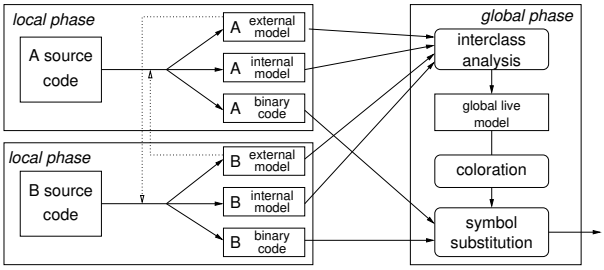

Figure 3: Global Phase

3.2 Global Phase

The global phase is divided into three stages: (i) type analysis which determines the live code and the *live global model*, (ii) coloring, and (iii) symbol substitution in the binary code (Figure 3).

Type analysis is based on the internal and external models of all classes. With flow analysis, the internal model of properties are linked together by connecting their entries and exits in order to constitute a global constraint network. In polyvariant analysis [1], the internal models can be duplicated to take into account the call contexts. The live classes and their live attributes and methods are identified, as well as the information on the concrete types of the live call sites.

The coloring stage is performed once the live global model is obtained. A heuristic [20, 22] produces the values of the identifiers and the colors of the live classes, methods and attributes, as well as the size of the instances. Computing a coloring at link-time was first proposed by [20] but, to our knowledge, this has never been implemented.

The last stage substitutes values to symbols. Colors computed during the coloring stage are substituted to the corresponding symbols. For each late binding site, the symbol is replaced according to the polymorphism. On a monomorphic site, the symbol is replaced by the address of the single method: the result is a direct call. On a polymorphic site, the symbol is replaced by the address of a resolver. Resolvers are small link-time generated functions that select the correct method. With BTD implementation, resolvers only contain a select tree where leaves are static jumps to the correct function. With VFT implementation, resolvers only contain a jump to the required method in the function table.

4. BENCHMARKS

4.1 Description

The following benchmarks compare our compilation framework with a pure global compiler and a pure separate compiler. We also compare three implementation techniques: VFT with subobjects, VFT with coloring and BTD.

4.1.1 Languages and Compilers

g++ is the GNU C++ compiler from `gcc`, the GNU Compiler Collection. It uses the standard C++ implementation with VFT and subobjects. As a part of `gcc`, it generates executables written in machine language. Because our benchmark programs use multiple inheritance and late binding, the `virtual` keyword is used both with inheritance and method definition[4] and the instances are manipulated through pointers[5].

SmartEiffel is the GNU Eiffel compiler. It uses a fast flow independent type analysis and implements object-oriented mechanisms mainly with BTD. It compiles programs into C then uses a C compiler to build an executable program.

prmc is our compiler for Prm, an Eiffel subset toy language. It has two compilation modes for polymorphic late binding: either with VFT and coloring or with BTD. It implements attribute accesses with coloring and typechecks with subtyping table and coloring. As SmartEiffel, it compiles programs into C then uses a C compiler to build an executable program. Currently, `prmc` performs a simple RTA [2] type analysis (without flow analysis) and partially removes the dead code[6].

For SmartEiffel and `prmc`, we use `gcc` as a C compiler. Hence, the three object-oriented language compilers use the same back-end for machine code generation. Our benchmarks are performed on a Bi-Xeon CPU 1.80GHz. `gcc`, version 4.0.0, is used with two options: `-O2` and `-fomit-frame-pointer`. SmartEiffel, version 1.1, is additionally used with `-boost` and `-no_gc`. The 2.2 version of SmartEiffelgives the same results.

4.1.2 Programs

In order to ensure the relevance of the results, the programs of the benchmark must meet two requirements :

- The same programs must be available for each language.
- A program must focus on only one of the three object-oriented mechanism.

With a small script program, we generates an identical small program for each language. Each program is based on a repetition of actions of the same type on different receivers. It consists of three parts: class definitions, an initialization sequence and a loop.

- Classes are generated. In order to avoid some artifacts, five isomorphic class hierarchies are generated. Each class in the hierarchy (re-)defines methods and introduces a new attribute. The number of classes in each hierarchy is a parameter of the script.
- For each hierarchy, we define a simple array[7] where elements are statically typed by the root of the hierarchy. Then the array is filled with a random sequence

[4]In C++, the *virtual* keyword is used to avoid repeated inheritance, and late binding requires that methods are defined with the *virtual* keyword.

[5]C++ late binding also requires that receivers are either pointers (∗) or references (&).

[6]The granularity is the binary file, so the global phase only removes binary files that contain only dead functions and dead classes. Future versions will remove more dead code.

[7]g++, SmartEiffel and `prmc` handle native C arrays. We do not use the high-level collection of the language library (for instance the `Vector` class of the C++ standard template library) since their implementation are not equivalent in each language.

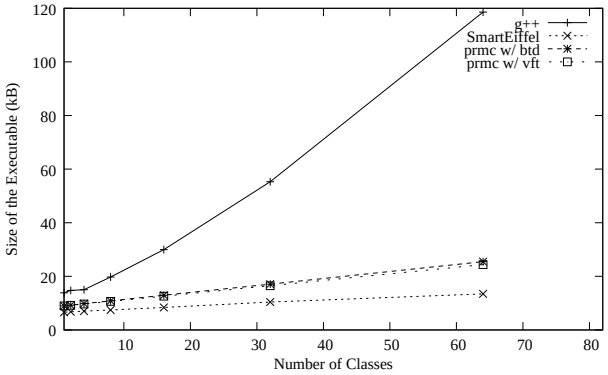

Figure 4: Size of Stripped Executables

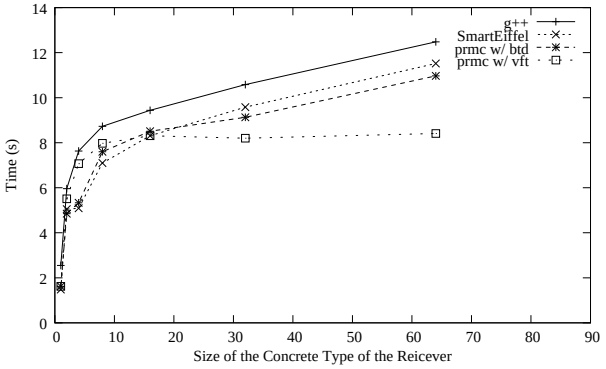

Figure 5: Late Binding

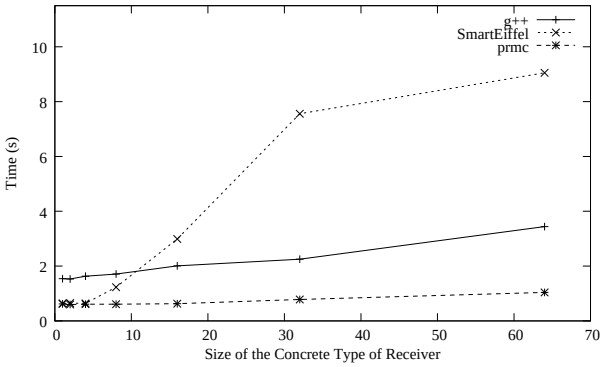

Figure 6: Attribute Access

of instances. For each language we generate the same random sequence.

- A loop that repeats actions on each element of the array. The generated code looks like:

```
for i from 0 to many do
       for j from 0 to arraylength do
              action1 on array1[i]
              action2 on array2[i]
              ...
              action5 on array5[i]
       end for
end for
```

  According to an option of the script, the action performed may be a method call, an attribute access or a cast. See the Appendix for an example of a program generated for C++.

For each benchmark, we measure the quantity of time needed by the program according to the size of the concrete type of the array items (i.e. the size of the hierarchy minus one as the arrays contain only subclasses). Experiments show that initialization time is negligible.

4.2 Results

First of all, we examine the size of the compiled programs. Figure 4 shows the size of stripped executables of simple generated programs. The programs contain class definitions in a binary hierarchy in single inheritance, the instantiation of these classes and a late binding site. The x range is the number of classes and the y range is the size in kilobytes. As expected, C++ generates large binaries (because the size of VFT with subobjects is cubic). SmartEiffel generates the smallest binaries because `prmc` does not remove all the detected dead code. However, the difference between `prmc` with BTD and `prmc` with VFT is not significant (because the size of VFT with coloring and the size of BTD code are both quadratic).

Our second benchmark, Figure 5, tests late binding: the actions performed in the loop are simple method calls. SmartEiffel and `prmc` compile monomorphic call sites with a static direct call. C++ uses VFT even on monomorphic call sites but the processor seems to manage them efficiently. Comparing VFT with BTD, BTD is more efficient for oligomorphic call sites[8] because of the processor static jump prediction but VFT is more efficient for megamorphic[9] call sites because of the complexity ($O(1)$ for VFT and $O(\log(n))$ for BTD). As expected, SmartEiffel and `prmc` with BTD have the same performance. These results are conform with [26]. The C++ overhead is due to the subobject implementation and cache misses.

For the attribute access benchmark, Figure 6, we read and write attributes introduced in a superclass of the static type of the receiver. The coloring gives the best results to `prmc`. The C++ implementation with subobject is less efficient. The SmartEiffel case requires an explanation: on attribute access, if the classes of the concrete type store the attribute at the same offset, the implementation is a direct access as with coloring [25]. If not, the implementation is a call to a function that uses a select tree and gets the correct attribute. However, SmartEiffel global compilation is fully compatible with coloring, and a global compilation scheme with coloring should be, at least, as efficient as `prmc`.

Type checking performances are measured with a type casting benchmark (Figure 7). The programs try to downcast each object of the array into the first subclass of the root hierarchy. Since in this benchmark, each hierarchy is a binary hierarchy, there is a 50 per cent chance the cast suc-

[8] We say that a call site is *oligomorphic* when the concrete type of the receiver is small.
[9] We say that a call site is *megamorphic* when the concrete type of the receiver is big.

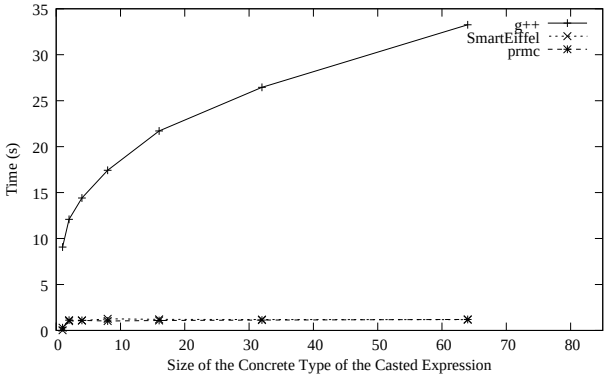

Figure 7: Type Downcast

cesses. We use the `dynamic_cast` construction with C++ and the `?=` construction with EIFFEL and `prmc`. Since the C++ type check implementation is complex and requires a function call, its performances are not good. SMARTEIFFEL and `prmc` both have good performances even if SMARTEIFFEL is better for oligomorphic sites and `prmc` better for megamorphic sites. However, contrary to late binding, the difference is small.

5. CONCLUSION

We present in this article a genuine separate compilation framework for statically typed object-oriented languages in multiple inheritance. It includes three global techniques of optimization and implementation: type analysis, coloring, and binary tree dispatch. Our proposition is a compromise between efficiency and modularity. It brings the efficiency of these global techniques without losing the advantages of separate compilation.

In comparison with classical separate compilation, the space and time reductions are significant. The type analysis detects monomorphic, oligomorphic and megamorphic method calls. Then the numerous monomorphic call sites are implemented by a direct and static call. The oligomorphic call sites can be resolved by an efficient binary tree dispatching. And the megamorphic call sites drop the expensive subobject VFT implementation thanks to coloring.

Comparing with pure global compilers, the performances stay honorable. However, from the point of view of efficiency, even if the quality of the type analysis is the same, SMARTEIFFEL and other global compilers keep a strong advantage with their code specialization techniques: method inlining, *customization* [4] or heterogeneous generic class compilation [19]. At least, like global compilers, our framework removes the justification of the two uses of the `virtual` keyword in C++ because the overhead of multiple inheritance (*virtual* inheritance) and monomorphic late binding (*virtual* functions) are mainly removed.

The question about shared libraries linked at load-time or dynamically loaded at run-time stays open. As a standard separate compilation framework with unresolved symbols, techniques with indirection tables are usable [15] but in our case, on the one hand the global analysis should be performed on load-time, and on the other hand, their efficiency is currently speculative.

The last open question is about the time overhead of the global phase (link), since it includes some static analysis and code generation. Our prototype, `prmc` is poorly implemented in a dynamic language. Hence, it can not be used to give a precise answer to the question. However, it seems that the overhead is quite small : less than 20% of the total compilation of a small program (local phases + global phase).

6. REFERENCES

[1] AGESEN, O. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, 1996.

[2] BACON, D. F., WEGMAN, M., AND ZADECK, K. Rapid type analysis for C++. Tech. rep., IBM Thomas J. Watson Research Center, 1996.

[3] BOUCHER, D. *Analyse et Optimisations Globales de Modules Compilés Séparément*. PhD thesis, Université de Montréal, 1999.

[4] CHAMBERS, C., AND UNGAR, D. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *Proc. OOPSLA'89* (New Orleans, 1989), ACM Press, pp. 146–160.

[5] COHEN, N. Type-extension type tests can be performed in constant time. *Programming languages and systems 13*, 4 (1991), 626–629.

[6] DIXON, R., MCKEE, T., SCHWEITZER, P., AND VAUGHAN, M. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89* (New Orleans, 1989), ACM Press.

[7] DRIESEN, K., AND HLZLE, U. The direct cost of virtual function calls in c++. In *Proc. OOPSLA'96* (1996), SIGPLAN Notices, 31(10), ACM Press, pp. 306–323.

[8] DUCOURNAU, R. Implementing statically typed object-oriented programming languages. Tech. Rep. 02-174, L.I.R.M.M., 2002.

[9] FERNANDEZ, M. F. Simple and effective link-time optimization of Modula-3 programs. In *SIGPLAN Conference on Programming Language Design and Implementation* (1995), pp. 103–115.

[10] GIL, J., AND ITAI, A. The complexity of type analysis of object oriented programs. In *Proc. ECOOP'98* (1998), E. Jul, Ed., LNCS 1445, Springer-Verlag, pp. 601–634.

[11] GOLDBERG, A., AND ROBSON, D. SMALLTALK*: the language and its implementation*. Addison-Wesley, Reading, MA, 1983.

[12] GOSLING, J. *The Java language specification*. Addison-Wesley, Boston, 2000.

[13] GROVE, D., AND CHAMBERS, C. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*, 6 (2001), 685–746.

[14] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP'91* (1991), P. America, Ed., LNCS 512, Springer-Verlag, pp. 21–38.

[15] LEVINE, J. R. *Linkers and Loaders*. Morgan-Kauffman, October 1999.

[16] LIPPMAN, S. *Inside the C++ Object Model*. Addison-Wesley, New York (NY), USA, 1996.

[17] Masini, G., Napoli, A., Colnet, D., Léonard, D., and Tombre, K. *Object-Oriented Languages*. Academic Press, London, 1991.

[18] Meyer, B. *Eiffel - The language*. Prentice-Hall, 1997.

[19] Odersky, M., and Wadler, P. Pizza into Java: Translating theory into practice. In *Proc. POPL'97* (1997), ACM Press, pp. 146–159.

[20] Pugh, W., and Weddell, G. Two-directional record layout for multiple inheritance. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'90)* (1990), ACM SIGPLAN Notices, 25(6), pp. 85–91.

[21] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading (MA), USA, 1986.

[22] Takhedmit, P. Coloration de classes et de propriétés : étude algorithmique et heuristique. Mémoire de dea, Université Montpellier II, 2003.

[23] Ungar, D., and Smith, R. SELF: The power of simplicity. In *Proc. OOPSLA'87* (Orlando, 1987), N. Meyrowitz, Ed., ACM Press, pp. 227–242.

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

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

[26] Zendra, O., and Driesen, K. Stress-testing Control Structures for Dynamic Dispatch in Java. In *2nd Java Virtual Machine Research and Technology Symposium (JVM 2002), San Francisco, California, USA* (Aug. 2002), Usenix — The Advanced Computing Systems Association, pp. 105–118.

APPENDIX

A. BENCHMARK PROGRAM

This appendix contains a part of the late binding program for 8 live classes generated for C++. We consider only two hierarchies instead of five in the real benchmarks.

```
/******************
 * file classes.h *
 ******************/
// First binary hierarchy of 9 classes
class C1 { // root class (is dead)
public:
        virtual void foo0(int x);
        virtual void foo1(int x);
        int a1;
};
class C2 : public virtual C1{
public:
        virtual void foo0(int x);
        virtual void foo1(int x);
        int a2;
};
class C3 : public virtual C1{
public:
        virtual void foo0(int x);
        virtual void foo1(int x);
        int a3;
};
class C4 : public virtual C2{
public:
        virtual void foo0(int x);
        virtual void foo1(int x);
        int a4;
};
// etc.

// Second binary hierarchy of 9 classes
class C10 { // root class (is dead)
public:
        virtual void foo0(int x);
        virtual void foo1(int x);
        int a1;
};
// etc.
```

```
/********************
 * file classes.cpp *
 ********************/
#include "classes.h"
// Since the following methods will be called
// with a negative argument, they will do nothing
void C1::foo0(int x) { if (x > 0) a1 = x; }
void C1::foo1(int x) { if (x > 0) a1 = x; }
void C2::foo0(int x) { if (x > 0) a2 = x; }
void C2::foo1(int x) { if (x > 0) a2 = x; }
void C3::foo0(int x) { if (x > 0) a3 = x; }
void C3::foo1(int x) { if (x > 0) a3 = x; }
void C4::foo0(int x) { if (x > 0) a4 = x; }
void C4::foo1(int x) { if (x > 0) a4 = x; }
// etc.
```

```
/************************
 * file message_send.cpp *
 ************************/
#include <stdlib.h>
#include "classes.h"
int actions(C1 ** tab0, C10 ** tab1)
{
        // Long loop
        for(int n=0; n<125000; n++)
        {
                for(int i=0; i<400; i++)
                {
                        // late binding
                        tab0[i]->foo0(-1);
                        tab1[i]->foo1(-2);
                }
         }
}

// Initialization
int main(void)
{
        // Allocations
        C1 ** tab0;
        C10 ** tab1;
        // Allocate the first array
        tab0 = (C1 **)calloc(400, sizeof(C1*));
        // Fill the array with random instances
        // (from C2 to C9) for the first array
        tab0[0] = new C8();
        tab0[1] = new C2();
        tab0[2] = new C5();
        tab0[3] = new C2();
        tab0[4] = new C8();
        tab0[5] = new C2();
        tab0[6] = new C7();
        tab0[7] = new C5();
        tab0[8] = new C3();
        tab0[9] = new C5();
        // etc.

        // Run the actions
        action(tab0, tab1)
        return 0
}
```