Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler.

Olivier ZENDRA, Dominique COLNET and Suzanne COLLIN
E-mail: {zendra, colnet, collin}@loria.fr

Centre de Recherche en Informatique de Nancy
Campus Scientifique, Bâtiment LORIA,
54506 Vandoeuvre-lès-Nancy BP 239 Cedex
France

Abstract

SmallEiffel is an Eiffel compiler which uses a fast simple type inference mechanism to remove most late binding calls, replacing them by static bindings. Starting from the system's entry point, it compiles only statically living code, which saves compiling and then removing dead code. As the whole system is analyzed at compile time, multiple inheritance and genericity do not cause any overhead.

SmallEiffel features a coding scheme which eliminates the need for virtual function tables. Dynamic dispatch is implemented without any array access but uses a simple static binary branch code. We show that this implementation makes it possible to use modern hardware very efficiently. It also allows us to inline more calls even when dynamic dispatch is required. Some more dispatch sites are removed after the type inference algorithm has been performed, if the different branches of a dispatch site lead to the same code.

The advantage of this approach is that it greatly speeds up execution time and considerably decreases the amount of generated code.

1 Introduction

Object-oriented programming has become a major trend both in computer science and computer engineering. Indeed, heavy use of inheritance and dynamically-bound messages is likely to make code more extensible and reusable. However, some concern still exists about performance of object-oriented systems, especially because of dynamic dispatch.

In order to reach performance similar to that of traditional languages like C, object-oriented systems must implement message dispatch efficiently. Recent work on type inference is a first step in this direction, since it allows the replacement of many polymorphic call sites by monomorphic direct call sites. The method used to implement remaining polymorphic call sites also has a significant impact on speed. Furthermore, optimization techniques for object-oriented programs must be fast enough not to be used only for application delivery. Fast compilers are needed to match the incremental nature of object-oriented languages.

In this study, we present the results obtained during the SmallEiffel project, started three years ago. This real-size project brought us to the somewhat surprising following conclusion: separate compilation doesn't outperform whole system compilation, even for incremental development. We indeed claim that global system type inference combined with a powerful implementation of dynamic dispatch can be used for incremental development.

The SmallEiffel compiler was designed to validate existing assumptions and offer new ideas. A powerful type inference algorithm was implemented and validated on a large-scale (50,000 lines) project, as well as some more specialized benchmarks. The method consists in completely removing all Virtual Function Tables (VFTs) or similar structures, replacing them by simple but efficient type tests. Since all indirect calls have been replaced by direct ones, this implementation of dispatch sites allows code inlining even inside polymorphic call sites.

The method described here is not limited to Eiffel [Mey94], but can also be used for any class-based language [MNC$^+$91] without dynamic class creation or modification: for example, it is possible to apply the same method to C++ [Str86] but not to Self [US87].

The remainder of this paper is organized as follows. Section 2 provides some background material on type inference and Virtual Function Tables (VFTs). Section 3 describes the method used to remove as many polymorphic call sites as possible and efficiently implement remaining dispatches. Results and the benchmarks they were obtained from are presented in section 4. Section 5 reviews related work and section 6 concludes.

2 Background

Object-oriented languages derive significant expressive power from *polymorphism*, the ability to use an object of any concrete type, as long as it conforms to the required abstract type.

The overall purpose of type inference applied to object-oriented languages is information extraction, type checking and application optimization. Ole Agesen's PhD thesis [Age96] contains a complete survey of type inference systems. Reviewed systems range from purely theoretical ones [VHU92] to systems in regular use by a large community [Mil78], with partially implemented systems [Suz81, ST84] and systems implemented on small languages [GJ90, PS91, PS92].

While much research has been done on type inference [CF91, APS93, PC94, AH95, DGC95, EST95], with interesting results, common production compilers don't seem to take advantage of powerful type inference algorithms. Indeed, implementations of such algorithms are sometimes not fast enough for incremental development, especially since they require knowledge of the whole system, which often prevents separate compilation.

Most previously published dispatch techniques have been studied by Driesen et al. [DHV95]. There are two major kinds of dispatch techniques: dynamic and static. Dynamic dispatch techniques rely on run time or profile-guided information [HCU91, AH96]. They consist of various forms of caching at run time [DS84, UP87]. Static dispatch techniques precompute their data and code structure at compile time in order to minimize the work performed at runtime. Typically, the dispatch code retrieves the address of the target function by indexing into a table — the VFT — and performs an indirect jump to that address. There are variants of the VFT method (see section 4 in [DHV95]). Most of them use at least one array access and a function pointer.

Previous papers [DH96] have shown that the VFT mechanism doesn't schedule well on modern processors, since unpredictable conditional branches as well as indirect branches break control flow and thus are expensive on superscalar architectures. Another drawback of use of VFTs is that a polymorphic call site can never be inlined, because function pointers are required in all cases, including simple operations like attribute accesses. For example, an access to an attribute value via a VFT implies an access function definition. The method studied in this paper allows us to inline monomorphic and polymorphic call sites similarly. Thus, access functions do not need to be defined.

Since for application delivery, executable size is of some importance, dead-code elimination [Dha91, KRS94] is also a factor to be considered.

3 Method

This section describes the whole compilation process we used: type inference, implementation of dispatch, inlining, extra polymorphic call sites removal and finally the recompilation strategy.

3.1 Type inference

The first stage of this method consists in removing as many polymorphic sites as possible, replacing them by direct calls. This is done by a type inference algorithm, previously described in [CCZ97]. It can be considered as the combination of RTA (Rapid Type Analysis [BS96]) and customization [CU89] algorithms. SmallEiffel's type inference algorithm can be qualified as *polyvariant* (which means each feature may be analyzed multiple times) and *flow insensitive* (no data flow analysis).

Briefly, our algorithm builds and analyzes the call graph of the whole system and computes the set of all possible concrete types at run time. Dead code is never reached and thus never compiled, which avoids the cost of unnecessary compilation followed by code removal. Each living routine is duplicated and customized according to the concrete type of the target. Genericity as well as multiple inheritance are taken into account by our algorithm. This analysis doesn't imply any run time overhead since it is done during compilation.

In previous experiments [CCZ97], we showed this type inference mechanism was able to replace many occurrences of polymorphic sites by direct calls: more than 80% is a usual score. As the whole system is explored, efficient dead code removal is performed, since only living methods of a class are customized. This results in a significant speedup of the application execution and compilation times. For example, compiling SmallEiffel itself — which represents about 50,000 lines of Eiffel code — takes less than 10 seconds on a Pentium Pro (200MHz, 32Mb RAM).

Even when a polymorphic call (multiple possible target types) cannot be replaced by a direct call (one target), the number of possible target types is reduced.

No data flow analysis is performed in the current version. However, since class instantiation information seems to be more important than the flow-based information [BS96], the cost of such an analysis appears to be too high for the expected gain.

Obviously, the 100% limit cannot be reached for all programs whatever the quality of the inference algorithm used: a simple array filled with different mixed objects typed in is enough to break down any type inference system.

Most programs are thus likely to contain polymorphic sites, even after the best type inference analysis. When considering all the benchmarks of [DMM96], one may notice that polymorphic sites are called 26 times more frequently than monomorphic sites. Therefore, optimization of remaining polymorphic sites is crucial.

3.2 Removing Virtual Function Tables

Removing Virtual Function Tables (VFTs) is the solution we have chosen to optimize remaining dispatch sites.

Indeed, VFTs is the most common way to implement dynamic dispatch. However, compilers as well as modern architectures require a substantial amount of control flow information to fully exploit the hardware. As pointed out in [DH96, DMM96], VFT dispatch sequences do not schedule well on superscalar processors because they cause frequent unknown control transfers.

The method we use to implement dynamic dispatch is closely linked to Polymorphic Inline Caches (PICs) [HCU91]. PICs extend dynamic inline caching [DS84] to handle polymorphic call sites, by caching for a given polymorphic call site all lookup results, instead of only the last one. Shared PICs consists in using the same PIC for several call sites, caching for a given message name all receiver types known so far.

Our implementation of dynamic dispatch is a statically computed variation of shared PICs. As suggested in [HCU91], we use binary search to check the receiver type. To our knowledge, this method has never been tested before in Eiffel, at least on such a large scale.

Removing VFTs implies modifying the run time object structure. Generally each object structure contains a pointer to the appropriate VFT. SmallEiffel's method replaces this pointer by a statically computed integer value to identify the corresponding concrete type. This integer ID allows us to explicitly test the dynamic type of the target.

With VFT-like methods, a dispatch site is mapped as at least one array access to get a function pointer, followed by the call to this function. Our method is a direct call to a specialized dispatch function, which spares both array accesses and function pointer. There is thus no need for any array, and the associated table compaction burden [DHV95] becomes irrelevant. All the information used to discriminate concrete types as well as selectors is in the code area. Obviously, direct function calls are also faster than indirect ones.

Thus, one function is specially defined for each kind of polymorphic site: its definition is done for one given selector with a statically known set of concrete types. This scheme applies similarly to single and multiple inheritance, with no increase in complexity, unlike VFT-based methods [DHV95, DH96].

For example, assume `x.f` is a polymorphic site whose target has four possible concrete types T_A, T_B, T_C and T_D. A dispatch function is especially defined to handle both the selector `f` and the set $\{T_A,T_B,T_C,T_D\}$. In another polymorphic call site, the same dispatch function can be used only when both the selector and the concrete type set are exactly the same. Using a dispatch function instead of directly inlining the dispatch code allows an important factorization of code. Without such factorization, the total size of the generated code would be much more important, as we shall see later in this paper.

The body of the dispatch function contains static multi-branch selection code with hard-coded type ID numbers. To avoid costly sequential inspection code inside the dispatch function, the set of possible concrete IDs is sorted at compile time. Obviously, this method is possible because the type inference algorithm is applied to the whole system.

The body of the dispatch function is efficient binary branching code. Assuming ID of T_A, T_B, T_C and T_D are respectively 19, 12, 27 and 15, the following dispatch scheme is produced for our last `x.f` example:

```
if id_x ≤ 15 then
      if id_x ≤ 12 then f_B(x)
            else f_D(x)
      endif
   else
      if id_x ≤ 19 then f_A(x)
            else f_C(x)
      endif
endif
```

Obviously $f_A(x)$, $f_B(x)$, $f_C(x)$ and $f_D(x)$ are not always function calls. As the concrete dynamic type of the target is known inside each selected branch, the corresponding behavior may be inlined.

3.3 Inlining

Because of the efficient type inference algorithm and also the way we implement dynamic dispatch, a lot of inlining sites are detected. One must keep in mind that classical VFT-based methods are an obstacle to inlining, since the compiler doesn't know which function will be called.

Most of SmallEiffel's inlining patterns are presented in this section. The label given to each inlining pattern will be used in some figures of the result section.

ARI Attribute Reader Inlining is done when the function is a simple attribute access: the function has only one instruction which returns an attribute. All call sites of such a function are inlined and there is no need to define the function. Monomorphic sites are directly inlined, whereas for polymorphic sites, inlining occurs in the branch of the corresponding type ID. As a consequence, attribute reader functions entail no overhead at all.

AWI Attribute Writer Inlining is performed when the procedure is only defined to set an attribute with the value passed as an argument. Such a procedure has only one argument with only one instruction and the argument is used to write an attribute. As for ARI, all sites are inlined and attribute writer functions do not entail any overhead.

DRI Direct Relay Inlining is used when the routine body has only one instruction to relay another routine directly applied to the same target. Such a case occurs when someone wants to alias a routine. For example, assume we have in class `COLLECTION[X]` the following Eiffel definition:

```
push(element: X) is
   do
      Current.add_first(element);
   end;
```

As `add_first` is called with the same target as `push` (Eiffel `Current` is equivalent to `this` in C++ or to `self` in Smalltalk [GR83]), the `push` procedure is inlined. All direct relay sites are inlined, and this kind of aliasing doesn't create any overhead or relay function definition.

DRI is also applied when arguments of the relayed call are statically computable. For example, this routine from class `STRING` of SmallEiffel's standard library is also inlined:

```
first: CHARACTER is
   do
      Result := Current.item(1);
   end;
```

Calling `first` has the same cost as calling directly `item(1)`. Thus improving readability does not cost anything.

DARI Direct Attribute Relay Inlining uses the fact that if the concrete type of `Current` is known, the concrete types of its attributes may be known as well. This is the case when the three following conditions hold. First, the routine has only one instruction which is a call. Second, the target of this call is an attribute of `Current`. Third, this call is monomorphic (the target attribute has only one possible concrete type).

For example, the function `item` of class `STRING` is defined as follows in SmallEiffel's standard library:

```
item(index: INTEGER): CHARACTER is
   do
      Result := storage.item(index - 1);
   end;
```

The call to `item` is monomorphic since attribute `storage` has only one possible concrete type (it is always an array of characters). As for previous inlinings, DARI sites are inlined and this kind of relay routine has no overhead.

PRI Predictable Result Inlining is used when the result of a function has a value known at compile time. The result may be any statically computable expression including nested function calls. For example, the following functions are both inlined:

```
two: INTEGER is
   do
      Result := 1 + 1;
   end;
three: INTEGER is
   do
      Result := 1 + Current.two;
   end;
```

The previous examples are trivial, but one must keep in mind that each living routine is customized for each concrete living type [CCZ97]. Thus, the call to `two` — which is currently statically computable — may be redefined in a subclass. The new definition will be considered separately and may be statically computable too.

RCI Result is Current Inlining is done when a function body has only one instruction to return the `Current` value. The following function is thus inlined:

```
foo(bar: Y): X is
   do
      Result := Current;
   end;
```

As we will show in the results section, this kind of strange function is not very common, though it exists.

EPI Empty Procedure Inlining is performed when a procedure has an empty body. Surprisingly, we have noticed that such a case is more frequent than expected (for example, the common Eiffel standard `do_nothing` is an EPI case).

OEI Other Eiffel Inlinings are performed by the SmallEiffel compiler. We do not describe those inlinings here because they are closely linked to the Eiffel language (e.g. inlining of pre-computable `once` functions). Unlike previous inlinings, OEI is not directly mappable to C++ or Java [JGS96].

3.4 Removing some more dispatch sites

Removing some more dispatch sites is still possible even after the type inference algorithm is finished. When all branches of the dispatch have exactly the same behavior — the code produced inside each branch has the same effect — no dispatch is needed. The polymorphic call site becomes a monomorphic one. Here is the classification of removed polymorphic sites.

ARR Attribute Read Removal is performed when the two following conditions hold. First, each branch of the dispatch is an ARI. Second, all attributes have the same common offset for all possible concrete types of the polymorphic site. Even when all attributes have the same name, it is important to check for the second condition because of multiple inheritance: one concrete type may inherit different attributes. For single inheritance languages, the second condition may be omitted.

AWR Attribute Write Removal is performed when the two following conditions hold. First, each possible concrete type of the polymorphic call is an AWI. Second, each attribute has the same common displacement. As for the previous removal pattern (ARR), the second condition may be omitted when the source language has no multiple inheritance.

DARR Direct Attribute Relay Removal is performed when all branches drive to the same relayed routine: all branches are DARI and the relayed routine is the same.

OER Other Eiffel Removals allow some other polymorphic calls to be removed. They are not described here because they are closely linked with the SmallEiffel interface with low level arrays. Like OEI, OER may not be applicable to C++ or Java.

3.5 Recompilation

Whole system analysis is likely to raise some concerns about compilation times. As we will show in section 4, the Eiffel to C translation from scratch is extremely fast. In order to avoid recompiling all C files, we use an amazingly simple and efficient process. It consists of the three following stages.

First, all C files and object files produced during the previous compilation are saved. Assume the saved C files are named $old_1.c$, $old_2.c$, ..., $old_n.c$. Next, the whole Eiffel source code is analyzed and all new C files are generated as $new_1.c$, $new_2.c$, ..., $new_n.c$. Note that each C file contains a bunch of code which is unrelated to the class hierarchy. Finally, for each pair ($new_i.c$, $old_i.c$), file contents are compared byte to byte. If the C file has not changed, the old object file is used, thus avoiding a C compilation.

The — very fast — Eiffel to C compilation is thus non-incremental, whereas the C compilation is. An advantage of this trivial technique is to avoid maintaining a project database with information that is implicitly in the C code. The Eiffel to C compiler work is thus reduced, which contributes to its speed.

4 Results

4.1 The SmallEiffel benchmark

The SmallEiffel benchmark is the most significant benchmark we have. SmallEiffel is fully bootstrapped in Eiffel and represents about 50,000 lines of Eiffel for about 300 classes. Everything is written in pure Eiffel: lexical analysis, parsing, semantic analysis, type inference, code generation and code optimization [AU77]. The standard library distributed with SmallEiffel (`ftp://ftp.loria.fr/pub/loria/genielog/SmallEiffel`) is the one used by the compiler itself. One may check that everything is full Eiffel even for very basic objects like `STRING` or `ARRAY`. The public `DICTIONARY` class is used for all symbol tables of SmallEiffel.

The compiler makes extensive use of dynamic dispatch to analyze Eiffel source code. For example, the abstract class `EXPRESSION` has no less than 48 concrete living types to implement all kinds of Eiffel expressions. Obviously, dynamic dispatch is used to select the appropriate behavior for each expression. The following table gives some other examples of the "Object-Orientedness" [DDG+96] of the SmallEiffel benchmark:

Concrete types	Abstract class
48	`EXPRESSION`
27	`CALL`
20	`TYPE`
17	`FEATURE`
16	`INSTRUCTION`
12	`NAME`
8	`ROUTINE`
8	`CONSTANT_ATTRIBUTE`
...	....

One must also keep in mind that Eiffel is a pure Object-Oriented language [Mey88] (ie. every routine is virtual, using C++ terminology).

In [CCZ97], we show some early results[1] related to the bootstrap process. We present here new benchmarks and analyses[2] that take into account more complete and optimized algorithms (especially those described in sections 3.3 and 3.4).

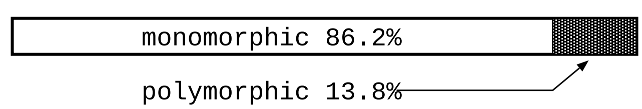
As SmallEiffel analyzes the whole system to produce code, one may be worried about compilation time. Figure 1 gives a survey on various architectures of the bootstrap process: SmallEiffel compiles itself with all previously described optimizations enabled. The total compilation time is divided in two parts. The upper part gives the time to produce the C code from the Eiffel source starting from scratch including the comparison of new and old C files. The middle part gives the time to produce the executable file from C code using `gcc -O6`.

This figure shows that SmallEiffel's Eiffel to C compilation is extremely fast: for example it takes less than 10 seconds on a Pentium Pro 200 to translate 50,000 lines of Eiffel into 65,000 lines of C code. This first stage — which contains all the previously described techniques — is about 10 to 20 times faster than the translation from C code to executable.

Total time from scratch to executable is only 2 minutes on a Pentium Pro 200. Furthermore, when one does not compile from scratch, only some C files need to be recompiled. Thus, with a minor change in a program of 50,000 lines of Eiffel, it takes only about 15 seconds to build the new executable on a Pentium Pro 200. This demonstrates that whole system analysis can be used for incremental development.

Figure 1 also gives the size of the executable file produced (the SmallEiffel compiler). Compared with other commercial Eiffel compilers or with gcc, the SmallEiffel executable is small (655Kb on a Pentium Pro 200). This demonstrates that extensive inlining and method customization combined with an efficient dead code removal do not increase the size of executables.

For the whole compiler, the type inference analysis final score is very good. There are 24975 monomorphic call sites and only 3983 polymorphic call sites:



As seen previously, some polymorphic sites are removed because each branch would produce the same code (ARR, AWR, DARR, OER). The gain is 1098 polymorphic sites.

[1] which come from the first public version of SmallEiffel, numbered `-0.99`

[2] obtained with the 14th version, named SmallEiffel `-0.86`

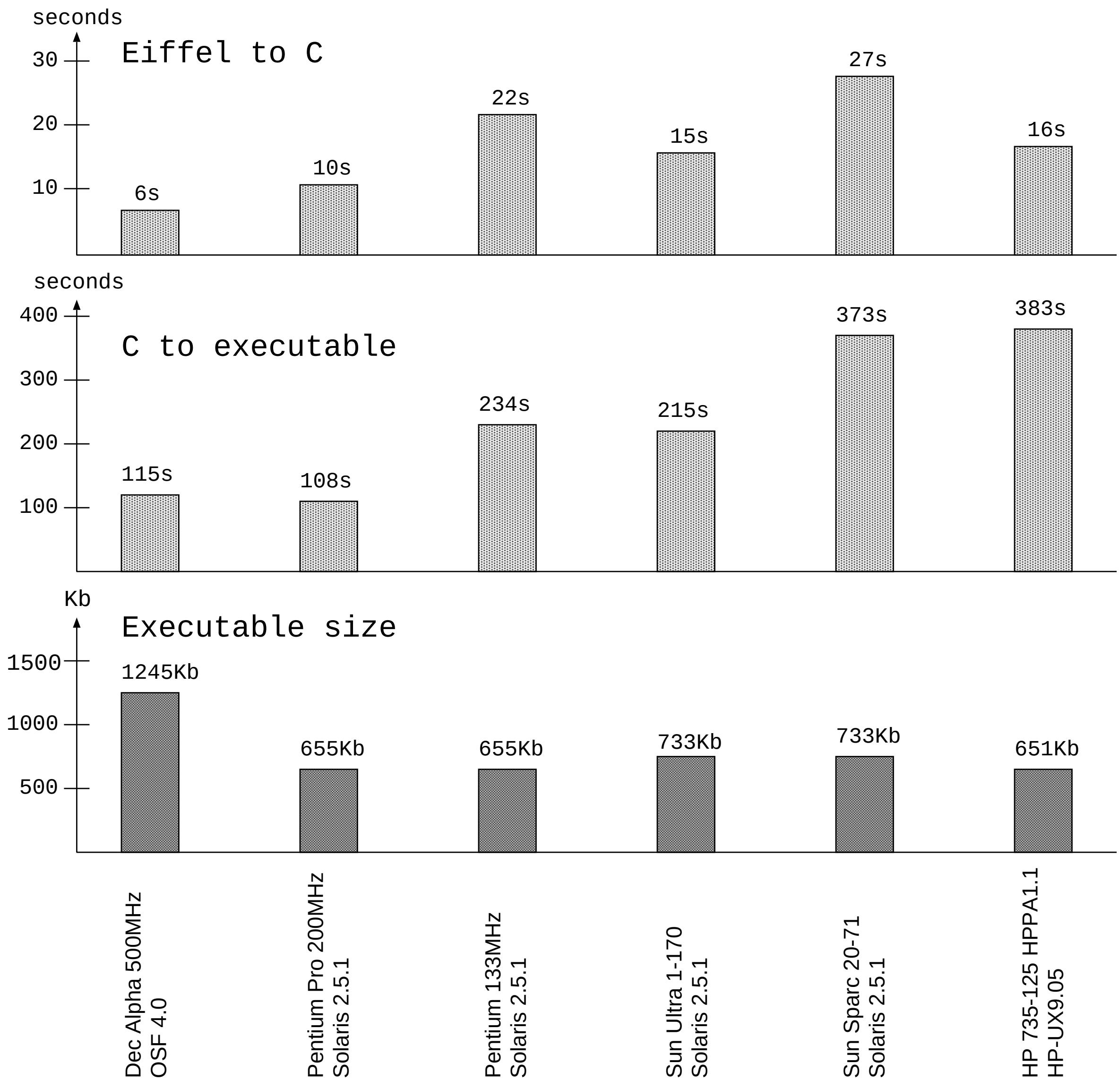
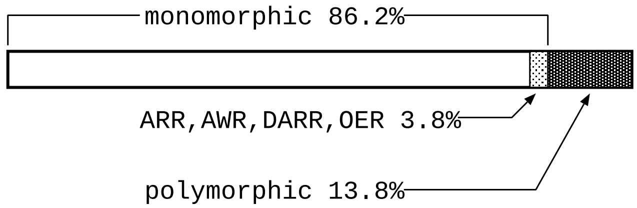
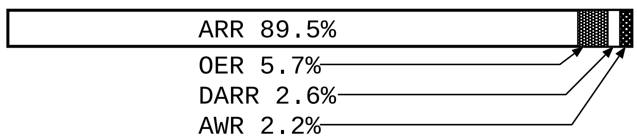


Figure 1: Bootstrapping SmallEiffel on various architectures.

The upper graph gives the total time from scratch to obtain 65,000 lines of optimized C from 50,000 lines of Eiffel. The middle graph shows the total time from C code to executable (link time is included). The lower graph gives the size of stand-alone executables.

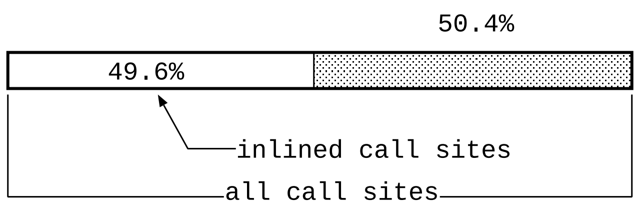


As those removals occur after the type inference algorithm is done, this benefit of 3.8% should not be neglected. Looking more closely at ARR, AWR, DARR and OER indicates that most polymorphic sites removed come from attribute read (ARR):

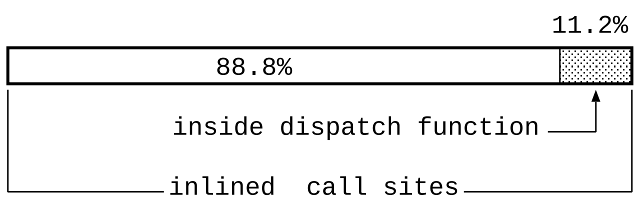


This information is likely to be of interest for other compilers writers.

The total number of inlined sites is 14376 for a total of 28958 sites:

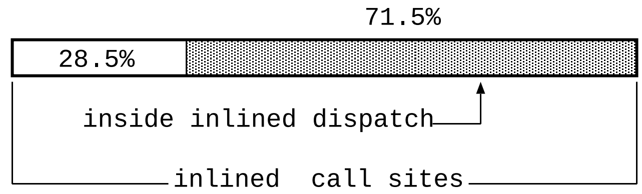


As seen previously, inlining may occur for a monomorphic site as well as inside a multi-branch dispatch function. For a total of 14,376 inlined sites, 1,603 are inside dispatch functions:



For 3,983 polymorphic call sites, there are 199 dispatch functions: the average number of calls per function is thus about 20. Consequently, the impact of one inlining inside a dispatch function may be considered 20 times more important than that of an inlined monomorphic call site.

Assuming that dispatch functions themselves are inlined gives the following ratio:



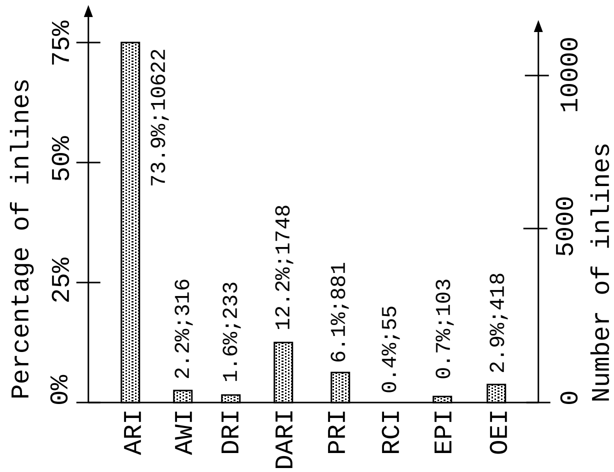
This kind of inlining may seem interesting to avoid the calls to the dispatch functions. However, these functions represent about 8,000 lines of C code, among a total 65,000 lines. Since each of them is called 20 times on average, inlining them would produce 160,000 lines, leading to a total program size of 217,000 lines (65,000+19×8,000). This would represent a threefold increase in code size.

Furthermore, since the call to a dispatch function is a direct one, it does not break control flow. The speedup we may expect from such inlinings seems thus limited. This, combined to the important code size increase, lead us not to implement these dispatch function inlinings. One must notice that the back-end C compiler remains free to perform such inlinings [3].

The following figure presents the distribution of the previously described inlining schemes regardless of their position (inside or outside dispatch functions).

Attribute Read Inlining (ARI) is by far the most common one. This is not surprising at all because accessing attributes is a very common operation. Furthermore, no attribute access function is ever defined. As described previously, all call sites to such a function are always inlined. As a consequence, there is no run time penalty when reading an attribute through an access function: maximum performance and encapsulation can be achieved simultaneously.

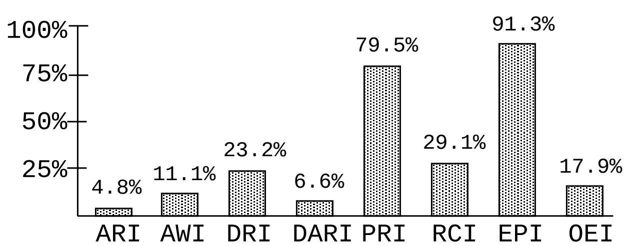
[3] For example, we have checked that gcc version 2.7.2 did inline small dispatch functions, as expected.



Conversely, the VFT method implementation of polymorphic call sites requires function pointers, which prevent any polymorphic function from being inlined.

Direct Attribute Relay Inlining (DARI) and Predictable Result Inlining (PRI) also represent a non-negligible part of inlinings. After looking at the generated code, we noticed that DARI often comes from encapsulation. Indeed, an encapsulating class often serves as a simple relay to services provided by private attributes.

An explanation of the importance of PRI can be deduced from the following figure, which shows — for each inlining scheme — the ratio of inlinings inside dispatch functions to the total number of inlinings.



Inlining inside dispatch functions

Most PRI are inside dispatch functions. A closer examination of the generated code reveals that this phenomenon is linked to a common use of inheritance in object-oriented programming: a virtual function in some abstract class is implemented with a constant result in most of its concrete subtypes. For example, function `number_of_wheels` of class `VEHICLE` has the constant value 4 in subclass `CAR` and 2 in subclass `MOTORCYCLE`.

The importance of Empty Procedure Inlinings (EPI) inside dispatch functions results from a similar cause. It is indeed not uncommon to redefine a selector `do_something` with an empty function body, or, conversely, to have an empty default behavior which is redefined to actually do something in only a few subtypes.

4.2 Dynamic Dispatch and VFT overhead

In order to quantify the impact of binary branch code compared with VFTs, we ran a simple benchmark written both in Eiffel and in C++. It only comprises one polymorphic dispatch site, embedded in a long-time loop. There is one abstract class `X` with three concrete subclasses `A`, `B` and `C`. The virtual abstract function `do_it` of `X` is redefined differently for each concrete class. Redefinition in each class is chosen to avoid any possibility of removal (section 3.4) by SmallEiffel.

Consequently, the following benchmark still has one remaining dispatch call site after type inference has been performed, which allows us to actually measure the cost of the dispatch method. Here is the loop given in C++:

```
{X *x;
 A *a = new A();
 B *b = new B();
 C *c = new C();
 int i;
 for (i = 100000000; i != 0; i--) {
    switch (i % 3) {
    case 0 : x = a; break;
    case 1 : x = b; break;
    default: x = c;
    }
   x->do_it(); //  The Polymorphic Call Site
  }
};
```

Since it rotates between the three branches, this benchmark prevents any target prediction.

This C++ version (g++) implements the dispatch with a VFT, whereas SmallEiffel standard generated C code uses a multi-branch dispatch function. In order to ascertain the only difference we could notice came from the dispatch implementation, we added a third version. It was obtained by manually modifying the C code produced by SmallEiffel, to replace the dispatch call site by a VFT call written in C, everything else remaining unchanged.

Figure 2 shows the execution times of those three benchmarks on various architectures. Of course, our purpose is not to compare processors, but to show that the results are not linked to a specific architecture.

The unmodified C code produced by SmallEiffel is always the fastest one. On the architectures we tested on,

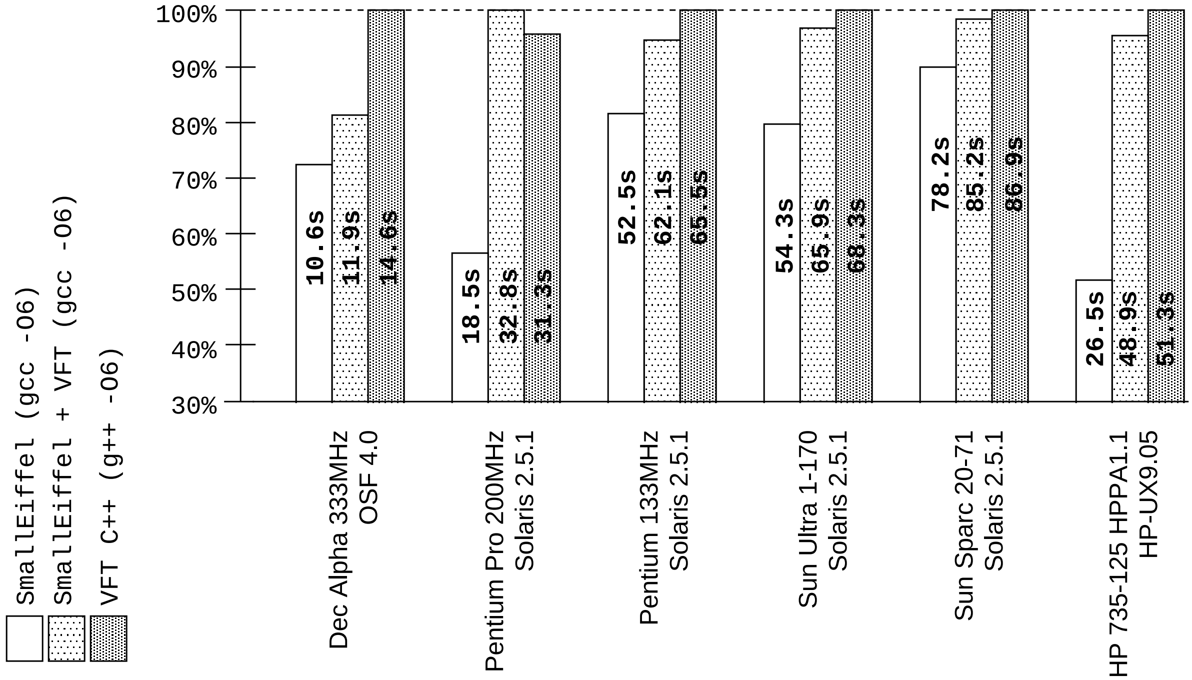


Figure 2: Execution time comparison for a 3 branch unpredictable call site: VFT vs. multi-branch.

SmallEiffel is in the worst case 10% faster than the C++ version, and 48% faster in the best case. This clearly shows that SmallEiffel's dispatch implementation is better for polymorphic call sites with few possibilities. Furthermore, times obtained with C++ and the modified SmallEiffel C code are consistent. Since they both implement dispatches with VFTs, this seems to confirm that VFT implementation of dynamic dispatch is less efficient than the method presented in this paper.

The VFT dispatch method has a constant execution time regardless of the number of possible concrete types of a polymorphic call site. Conversely, SmallEiffel's method cost is not constant because the number of branches inside the dispatch function grows with the number of possible concrete types. As we have chosen a binary branching code to test among various possible concrete types, the number of branches is given by $log_2(N)$ where N is the number of possible concrete types. In order to compare execution time for a *megamorphic* call site — i.e. a polymorphic call site with many possible receivers [DHV95] — we have extended the previous benchmark to 50 subclasses. According to [AH96], a total of 17 possible concrete types may be considered as megamorphic. One may also note that the most polymorphic call site of SmallEiffel (50,000 lines) has 48 possible concrete types (class `EXPRESSION`).

The previously described class `X` now has 50 subclasses, which leads to a 50 branch dispatch. Figure 3 shows results obtained with a benchmark where the code rotates the receiver among the 50 possiblities, making the dispatch call unpredictable. Conversely, the benchmark used for figure 4 does not rotate the receiver among several types, thus always invoking the method on the same receiver, which makes the call predictable.

Figure 3 shows that, for an unpredictable megamorphic call site, the binary branching code method and the VFT method have similar results. Indeed, on three architectures, conditional code outperforms the VFT code (by 4% to 31%), whereas on the three others VFTs are faster (by 4% to 7%). Consequently, the binary branching code method appears to be quite scalable. Since all branches are equiprobable, profile guided techniques [AH96] or inline caching [DS84, UP87] do not seem to be able to improve results.

Figure 4 shows that results are even better on a predictable call site, since conditional code now outperforms the VFT code on four architectures.

A closer look at the assembly code generated for the binary branching code and for the VFT mechanism leads to some interesting observations. First, data dependen-

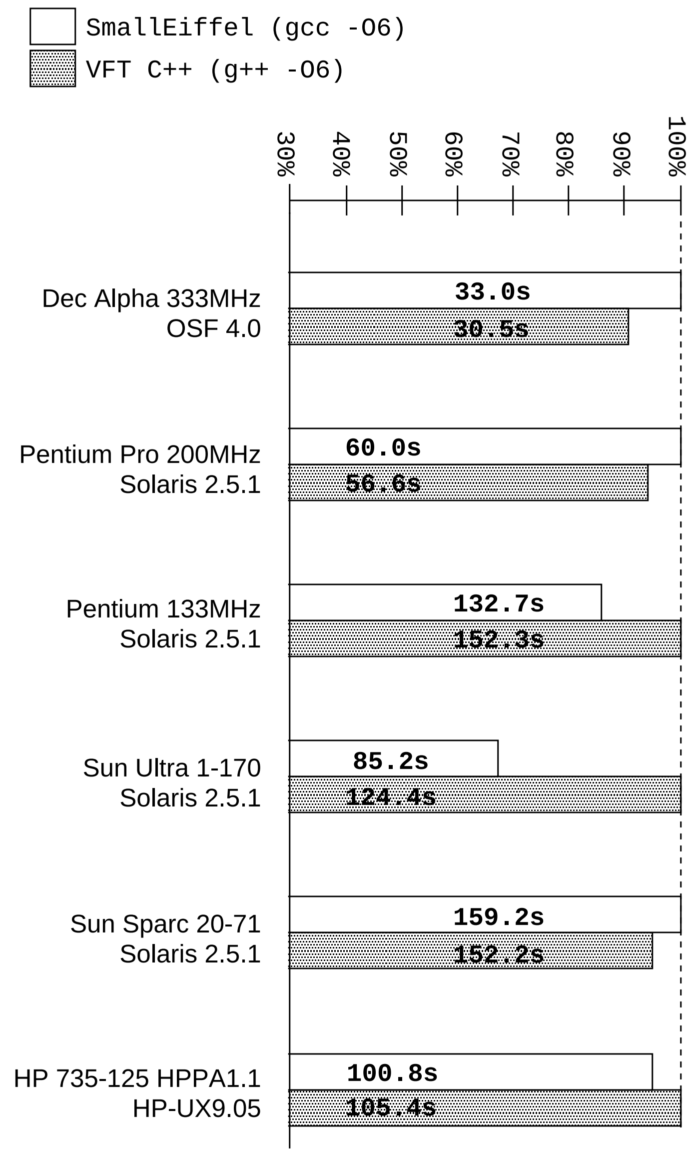


Figure 3: Execution time comparison for a 50 branch unpredictable megamorphic call site.

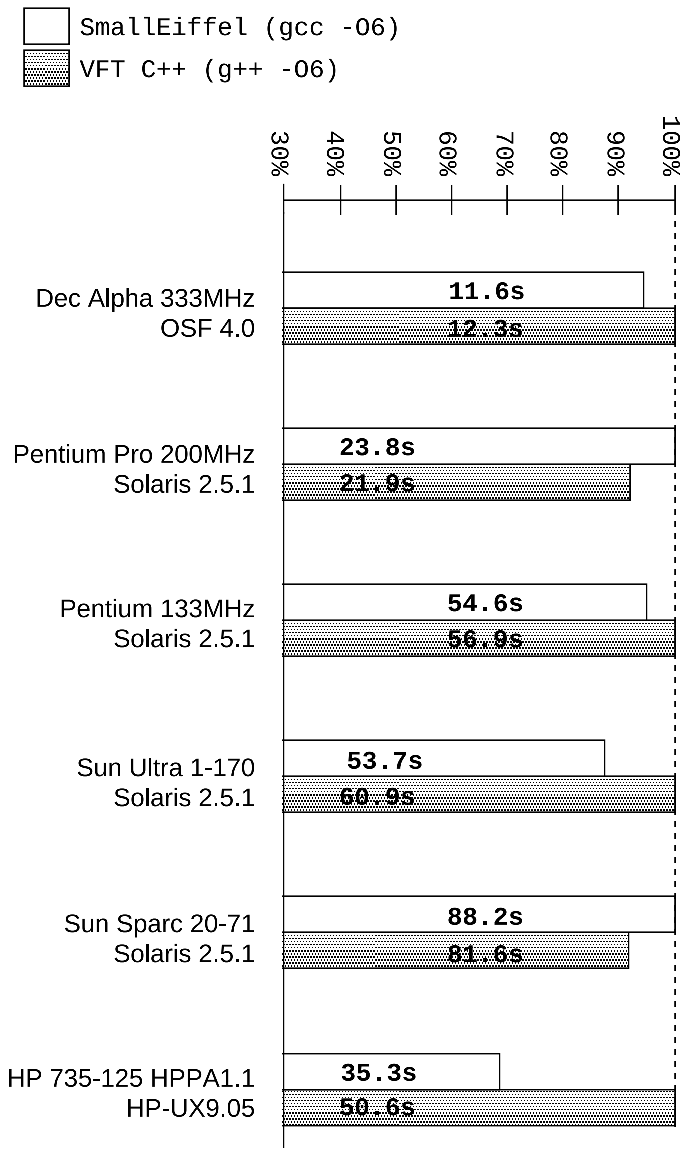


Figure 4: Execution time comparison for a 50 branch predictable call site.

cies are important in VFT code (see for example [DH96]), whereas they are less in binary branching code. However, in binary branching code, more control flow dependencies arise. The latter are likely not to be an important problem on architectures using Branch History Table (BHT) mechanisms, which will allow many conditional branches to be predicted. Indeed, it is a well-known property of polymorphism that generally, the receiver type at a polymorphic call site does not vary much. The BHT is thus used as a memory of the last receiver type, which may be considered as some kind of inline caching performed by the processor.

4.3 Other Comparisons with C++ and Eiffel Compilers

In this section we present the results obtained by Dietmar Wolz (Technische Universität Berlin) on an algorithm computing the colimit of a signature diagram (a construction from category theory which is useful for parameterization concepts in specification and programming languages and for graph transformations).

This algorithm was implemented both in Eiffel and in C++, and benchmarked with different compilers and/or libraries, on a Pentium 200 with 512 Kb cache and 192 Mb RAM, running LINUX (kernel 2.0.12, gcc 2.7.2).

The Eiffel program consists of 13 classes one of which — dynamic arrays, inheriting from `ARRAY[G]` — was adapted to the different compilers for performance optimization. The C++ program uses a similar structure and is based on the Standard Template Library (STL).

Results presented here can be found in their original form in the `comp.lang.eiffel` archive at Cardiff University (`http://www.cm.cf.ac.uk`).

Figure 5 was obtained by running both programs on a large, 2 million symbol diagram. Results include compilation and execution times, in the upper part of the graph, and memory footprint and executable file size in the lower part.

This benchmark is a striking illustration of the speed and efficiency of SmallEiffel's previously described method. First, the benchmark executable generated by SmallEiffel is among the very fastest ones, compared to those generated by other Eiffel compilers, or even C++ compilers.

Second, compilation times with SmallEiffel are the best, whether compared to those of other Eiffel or C++ compilers. Since SmallEiffel has been bootstrapped, its executable code itself was produced using the method described in this paper, which in turn confirms the efficiency of the method.

Figure 5 also clearly shows that SmallEiffel's method produces the smallest executables, despite code duplication — for type inference — and the use of explicit type tests — instead of function pointers — to implement dispatch.

To interpret memory results, one must take into account the fact that the C++ program uses hand-made memory management, but some leaks remain. Memory usage for C++ is thus not significant, but the comparison between the different Eiffel compilers is still valid.

Since SmallEiffel currently doesn't implement memory management, its executables were linked with a Boehm-Demers-Weiser garbage collection (GC) algorithm, publicly available at `ftp://ftp.parc.xerox.com/pub/gc`. This generic, heuristic algorithm was in no way tailored to SmallEiffel, and consequently has no knowledge about SmallEiffel's representation of objects. However, SmallEiffel's use of memory with garbage collection enabled is the second best, after the Eiffel/S compiler. This score is likely to be improved when a specific GC algorithm is implemented in SmallEiffel. Indeed, with garbage collection turned off, SmallEiffel makes the sparest use of memory.

5 Related work

Most previously published dispatch techniques have been studied by Driesen et al. [DHV95]. They show that mechanisms employing indirect branches (i.e., all table-based techniques) may not perform well on current and future processors since indirect branches entail multi-cycle pipeline stalls, unless a branch target buffer is present. Further study from Driesen and Hölzle [DH96] confirms these results for the specific case of VFTs in C++. Our study also shows the cost of VFTs, and proposes a method which seems to schedule better.

Calder and Grunwald [CG94a] studied profile-guided receiver class prediction to eliminate indirect function calls. They add extra explicit tests to skip the standard dispatch mechanism for the most common target types. Code generated by SmallEiffel is not unlike Calder and Gunwald's, since it consists in explicitly testing the receiver type and using direct function calls. However, an important difference is that explicit type testing is extended to all possible receiver types, and not only the most common ones. This allows a complete removal of any VFT-like standard dispatch mechanism.

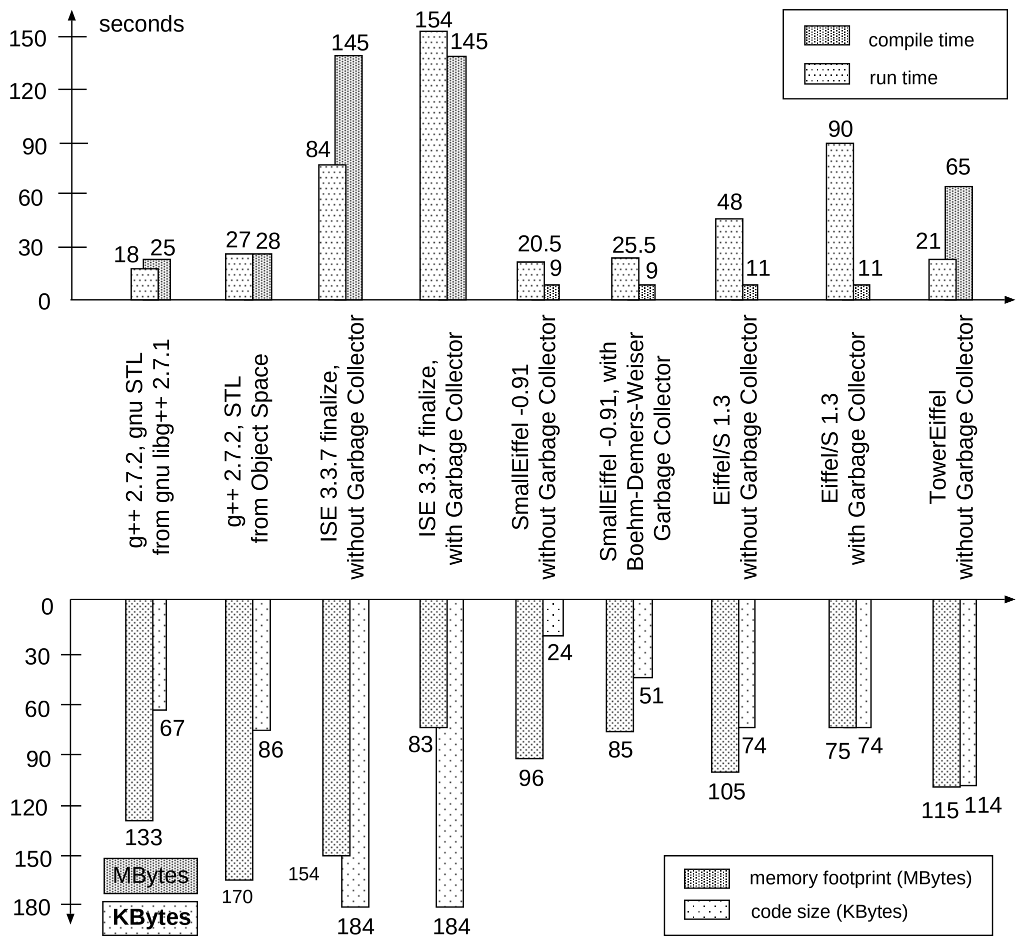


Figure 5: Overall performances of SmallEiffel vs. other compilers.

Using both profile-based optimization and class hierarchy analysis, Aigner and Hölzle [AH96] show how to improve efficiency of dynamic dispatch in C++. SmallEiffel uses a more complete type inference algorithm and implements remaining dispatch call sites without VFT at all. They also show that inlining barely increases code size, and that for most programs, the instruction cache miss ratio does not increase significantly. Our results are consistent with theirs both for executable sizes and execution time.

In both prior works, explicit type tests are inlined, whereas we factorize them in a dispatch function. Indeed, such inlining is not expensive, because only the most frequent receiver types are tested. As explained in section 4, this inlining is not reasonable when testing all receiver types explicitly. Profile-guided receiver class prediction could also be added to SmallEiffel to partially sort and optimize the binary branching code we use for dispatch.

In an early paper [Ros88], Rose proposes a *fat* table technique for dynamic dispatch in addition to the more classical VFT-like *thin* tables. Fat tables contain inlined code for small functions, instead of function pointers, thus allowing a branch directly into the table. According to Rose, fat tables can only contain very short methods which execute with no further transfer of control.

Dean et al. [DGC95] describe the Class Hierarchy Analysis (CHA) algorithm, used to remove polymorphic call sites. They show that CHA is fast enough to be supported in an interactive programming environment. They also indicate that despite the fact CHA needs information on the whole program, it can be adapted to allow incrementality. Our results also confirm this, thanks to the high speed of the SmallEiffel compiler.

Though SmallEiffel's type inference algorithm is more developed than CHA alone, Ole Agesen's Cartesian Product Algorithm (CPA, [Age95]) is an even more powerful one. Indeed, SmallEiffel currently doesn't perform any data flow analysis [CG94b] (neither intra- nor interprocedural). Using CPA in SmallEiffel would allow us to eliminate even more dispatch call sites, thus further increasing the speed of the generated code. Further studies are needed to precisely estimate the resulting speedup and extra cost.

Hölzle and Ungar [HU95] raise an interesting question: whether object-oriented languages need special hardware. They conclude that dispatch tests cannot easily be improved with special-purpose hardware, and that the most promising way to reduce dispatch overhead is via compiler optimizations. As an exemple, SmallEiffel's optimizations generally eliminate an average of 80% of dispatch call sites, generating only very basic instructions.

Bacon and Sweeney [BS96] investigate the ability of three types of static analyses to improve C++ programs by resolving virtual function calls and reducing compiled code size. Their best algorithm, RTA (including CHA as a core component), removes on average 71% of the virtual call sites, which is comparable to SmallEiffel's 80%. The better average score for our method may come from removals that occur after type inference (ARR, AWR, DARR and OER, of section 3.4)

6 Conclusions

We describe a method to implement polymorphism with a very high efficiency. Its first stage consists in a powerful type inference algorithm, working on the whole system, to replace polymorphic call sites by static ones. This algorithm reaches high average scores: 80% of polymorphic calls are resolved as monomorphic in our benchmarks.

Remaining dispatch sites are then coded efficiently, by boldly eliminating VFT-like tables and function pointers, and replacing them by a static binary tree of tests. These two stages allow the third one, inlining, to be performed extensively, both for static call sites and inside multibranch dynamic dispatch sites.

This method is implemented in a brand-new Eiffel compiler, started three years ago. To our knowledge, it is the first time such a method is applied to a full-scale project (a 50,000 line Eiffel compiler).

We demonstrate the validity of the method both in terms of speed and size of the generated code, even in the pathological cases of unpredictable or highly polymorphic (megamorphic) calls. We show the VFT overhead is avoided by this method, with no other extra cost.

The compilation technique we describe here for Eiffel may apply to any class-based language — even with genericity and multiple inheritance — but without dynamic class creation or modification. This is the case for C++ while it is not completely clear whether the method can be applied to Java, because of some more dynamic aspects. This issue seems worth investigating.

The major drawback of our method is that it requires knowledge of the whole system, which prevents separate compilation as well as the production of precompiled libraries. This may appear to be a problem for incremental development, but the most relevant aspect for the developer is compilation speed. Since the method allows very high compilation speed, as shown in section 4, we are convinced it is possible to easily integrate it in incremental production compilers.

The impossibility to deliver precompiled libraries seems to be a problem when their source code is confidential, e.g. for commercial reasons. However, a trivial solution consists in using encryption techniques. Of course this solution does not address the issue of run time shared libraries. Indeed, programs generated by SmallEiffel are able to interact with existing shared libraries (e.g made with C/C++ compilers), but it is currently impossible to create such libraries with SmallEiffel.

We think the technique can be easily adapted to produce precompiled libraries. However, precompiled library code is very likely to be less specific, hence less optimized, than the code of a standalone executable compiled from a complete system. We hope to address this in future work.

Our method uses only static information to improve code — especially dynamic dispatch — efficiency. Adding flow sensitivity to our compiler would increase performance of the generated code, since it would allow some more dispatch sites to be replaced by static calls. However, this may significantly increase compilation times.

As seen previously in section 3.4, when all branches of a dispatch function lead exactly to the same code, the corresponding polymorphic call sites are completely removed. It is also possible to further optimize the binary tree of tests by merging branches when only several of them lead to the same code. A coarse analysis of the generated dispatch trees seems to confirm that a lot of branches should be merged. In order to merge the largest number of dispatch functions properly, type ID assignment is a crucial issue. This is a rather complex problem that we have not adressed yet.

Other improvements may also come from heuristic methods, based on dynamic, run time information, such as inline caching.

Acknowledgments

We would like like to thank the anonymous reviewers for their helpful comments and suggestions, and Dietmar Wolz who allowed us to include his benchmark results. We are also grateful to Richard M. Washington, Philippe Ribet and Charles Després for proofreading early versions of this paper.

References

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 2–26. Springer-Verlag, 1995.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Department of Computer Science of Stanford University, Published by Sun Microsystem Laboratories (SMLI TR-96-52), 1996.

[AH95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *Proceedings of 10th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 91–107, 1995.

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Sciences*, pages 142–166. Springer-Verlag, 1996.

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF. Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Sciences*, pages 247–267. Springer-Verlag, 1993.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts, 1977.

[BS96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of 11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 324–341, 1996.

[CCZ97] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Joint Modular Languages Conference*, volume 1204 of *Lecture Notes in Computer Sciences*, pages 67–81. Springer-Verlag, 1997.

[CF91] Robert Cartwright and Mike Fagan. Soft Typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[CG94a] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead In C++ Programs. In *21st Annual ACM Symposium on the Principles of Programming Languages*, pages 397–408, January 1994.

[CG94b] Diane Corney and John Gough. Type Test Elimination using Typeflow Analysis. In *PLSA 1994 International Conference, Zurich*, volume 782 of *Lecture Notes in Computer Sciences*, pages 137–150. Springer-Verlag, 1994.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24 of *SIGPLAN notices*, pages 146–160, 1989.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proceedings of 11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 83–100, 1996.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 77–101. Springer-Verlag, 1995.

[DH96] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of 11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 306–323, 1996.

[Dha91] D.M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 253–282. Springer-Verlag, 1995.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *Proceedings of 11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 292–305, 1996.

[DS84] Peter L. Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. In *11th Annual ACM Symposium on the Principles of Programming Languages*, 1984.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound Polymorphic Type Inference for Objects. In *Proceedings of 10th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 169–184, 1995.

[GJ90] J. Graver and R. Johnson. A Type System for Smalltalk. In *17th Annual ACM Symposium on the Principles of Programming Languages*, pages 139–150, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Sciences*, pages 21–38. Springer-Verlag, 1991.

[HU95] Urs Hölzle and David Ungar. Do Object-Oriented Languages Need Special Hardware Support ? In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 283–302. Springer-Verlag, 1995.

[JGS96] Bill Joy James Gosling and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[KRS94] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–10000, 1994.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Mey94] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall, 1994.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences*, pages 348–375, 1978.

[MNC+91] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Object Oriented Languages*. Academic Press Limited, London, 1991.

[PC94] John Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented languages. In *Proceedings of 9th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'94)*, pages 324–340, 1994.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of 6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, pages 146–161, 1991.

[PS92] J. Palsberg and M.I. Schwartzbach. Safety Analysis Versus Type Inference for partial Types. *Information Processing Letters*, pages 175–180, 1992.

[Ros88] John R. Rose. Fast Dispatch Mechanisms for Stock Hardware. In *Proceedings of 3rd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88)*, pages 27–35, 1988.

[ST84] N. Suzuki and M. Terada. Creating Efficient System for Object-Oriented Languages. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 290–296, 1984.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Series in Computer Science, 1986.

[Suz81] N. Suzuki. Inferring Types in Smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199, 1981.

[UP87] David Ungar and David Patterson. What Price Smalltalk ? *IEEE Computer*, 20(1), 1987.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of 2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 227–241, 1987.

[VHU92] Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Sciences*, pages 237–250. Springer-Verlag, 1992.