SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **44**:565–592
Published online 26 November 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2174

Efficient compilation strategy for object-oriented languages under the closed-world assumption

Benoît Sonntag[1,‡] and Dominique Colnet[2,*,†§]

[1] *LSIIT, Université de Strasbourg, Illkirch, France*
[2] *LORIA, Université de Lorraine, Vandœuvre-lès-Nancy, France*

SUMMARY

Reaching the best level of runtime performance from a high-level, object-oriented language is often considered challenging if not unattainable. The closed-world assumption involves considering all of the source code of an application together at compile time. That assumption makes it possible to produce an efficient code. For instance, multiple inheritance can be implemented as efficiently as single inheritance. Our compilation strategy is the result of a prolonged project, tying together several compilation techniques: call graph analysis, dead code elimination, type flow analysis, code customization, implementation of dynamic dispatch, inlining, pointer optimization, switch optimization, objects layout, and so on. Merging all of these techniques into a global strategy appears to be quite problematic. Throughout the paper, two real-world compilers are used as benchmarks to provide measurements for compiler writers to evaluate the applicability of our approach. Type flow analysis is a fundamental aspect of our strategy to resolve method calls. We have extended type flow analysis to deal with the content of arrays, enabling us to process additional expressions and thus making it possible to obtain a true global analysis. Typically, more than 90% of method call sites are statically resolved. Our experience indicates that the closed-world assumption is suitable for numerous applications. Surprisingly, even library-defined control statements from dynamic languages are perfectly processed with our strategy. The Smalltalk `ifTrue:ifFalse:`, `whileTrue:`, `to:do:`, and so on are, for the very first time, perfectly translated. Copyright © 2012 John Wiley & Sons, Ltd.

Received 20 May 2010; Revised 5 November 2012; Accepted 5 November 2012

KEY WORDS: object-oriented; compiler; type flow analysis; dynamic dispatch; code customization; inlining

1. INTRODUCTION

When considering software organization, software reliability, or software modularity, object-oriented technology is currently the most popular way to meet all those requirements. On the other hand, there is substantially less consensus about the performance of object-oriented language. Many high-level, object-oriented languages do not have a good reputation for runtime performance. For many people, achieving the best runtime performance requires using C language or even assembly code.

Our results from working on the SmartEiffel and Lisaac compilers indicate that the previous assertion is far from obvious. When the application is very small, it is highly likely that C or assembly code is the best choice. As soon as the application is of medium or large size, we believe that a high-level, object-oriented language would be the best candidate, knowing that the compiler is a crucial

[*] Correspondence to: Dominique Colnet, LORIA, Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France.

[†] E-mail: Dominique.Colnet@loria.fr

[‡] Lisaac project leader, http://www.lisaac.org

[§] SmartEiffel project leader, http://SmartEiffel.loria.fr ingredient for success. Without claiming that object-oriented programming eliminates all of the disadvantages, we now believe that C could be omitted in many situations to improve productivity. This article presents our compilation strategy, which is suitable for several object-oriented languages. This efficient strategy is the result of many years of effort, first on the SmartEiffel compiler (formerly called SmallEiffel) and then on the Lisaac compiler. Both whose foundation in designs and implementations are deeply focused on ensuring runtime efficiency.

1.1. Context of our work: the SmartEiffel and Lisaac background

The work on the Lisaac language [1, 2] and compiler grew out of the work on the SmartEiffel compiler [3]. From the beginning, the SmartEiffel compiler has used global analysis. Keeping the global compilation technique, Lisaac has added type flow optimizations, hence improving code customization. Both compilers assume knowledge of all the source code during compilation. Adding new source code during runtime is forbidden. Thanks to the encouraging results of SmartEiffel, we decided to focus mostly on static optimizations, leveraging global program analysis. From the language point of view, SmartEiffel and Lisaac consider all types of data, such as booleans, integers, and pixels, as true objects. Furthermore, Lisaac, like Smalltalk [4] or Self [5, 6], also defines loops and conditional statements as part of the library. Block closures are also handled. Both languages are high-level, object-oriented languages featuring multiple inheritance. Being a prototype-based language, Lisaac is also a strongly typed language. Most of the work performed for SmartEiffel and Lisaac concerns high-level, object-oriented optimizations. The compilation strategy for inheritance and dynamic dispatch is a key point for object-oriented languages. Type flow analysis, mixed together with code customization and inlining, allows us to statically bind many method calls and is also used to predict a possible null pointer inside arrays. Low-level optimizations such as register allocation or loop unrolling are supposed to be applied after our high-level compilation strategy; however, to simplify the presentation, we use C as a target language.

1.2. Open-world versus closed-world assumption

Compiling under the closed-world assumption (CWA) means that the compiler is able to access the entire source code of the application to compile. Under CWA, not only are the classes and methods of the application considered but also the classes and methods of the libraries the application uses. Dead code, unused classes or unused methods, can be ignored and removed. Under CWA, no new code is supposed to be added after compilation time. Conversely, under the open-world assumption (OWA), new classes or subclasses may be added at any time. OWA is thus naturally associated with separate compilation, dynamic loading, and incremental development of code. As indicated in [7–11], it is well known that the main benefit of CWA is runtime efficiency. All method calls or attribute accesses can be customized prior to execution. Under CWA, multiple inheritance does not incur any overhead when compared with single inheritance. The code can be customized according to its real usage, and dynamic dispatch can be implemented using a hard-coded dispatch mechanism.

It is possible to use CWA for languages such as C++ and C# or for a large subset of Java (the most significant omissions are dynamic class loading and reflection as in [12]). Using CWA is more problematic for languages such as Smalltalk, Self, or Lisp because in these dynamic languages, new source code is likely to be a result of computation. Even if the language allows dynamic source code creation, many written applications do not use that feature. As an example, a Lisp application, which does not create new functions at runtime, can be compiled under CWA.

1.3. Global overview of our compilation process

Following the SmartEiffel compiler's strategy [3, 13–15], the Lisaac compiler also performs a global program analysis. Therefore, all the live code is considered to maximize type analysis for each method call site. Thanks to type flow analysis, the Lisaac compiler improves the type analysis of SmartEiffel by reducing the set of all possible types for each method call site as much as possible. Our compilation strategy features three major steps (Figure 1):

DOI: 10.1002/spe

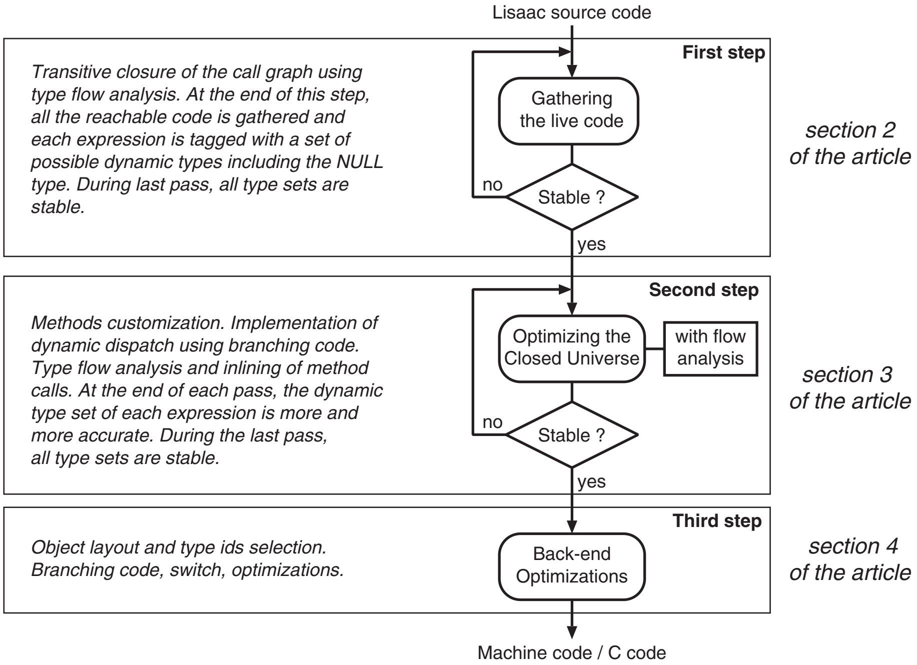

Lisaac source code
First step
Transitive closure of the call graph using type flow analysis. At the end of this step, all the reachable code is gathered and each expression is tagged with a set of possible dynamic types including the NULL type. During last pass, all type sets are stable.
Gathering the live code
no
Stable ?
yes
section 2 of the article
Second step
Methods customization. Implementation of dynamic dispatch using branching code. Type flow analysis and inlining of method calls. At the end of each pass, the dynamic type set of each expression is more and more accurate. During the last pass, all type sets are stable.
Optimizing the Closed Universe
with flow analysis
no
Stable ?
yes
section 3 of the article
Third step
Object layout and type ids selection. Branching code, switch, optimizations.
Back-end Optimizations
section 4 of the article
Machine code / C code

Figure 1. Global overview of our three steps compilation process.

First step. Gathering the live code. Using several passes, we aim to gather enough code to ensure that all the reachable code is included. Roughly, each pass of this step performs a transitive closure of the call graph. During each pass, each expression is tagged with a set of possible dynamic types. Each pass makes the type set of each expression bigger and bigger until a fixed point is reached. Adding a new dynamic type can make a new method attainable, which may eventually result in the addition of new types. Medium-size applications usually require 6 to 10 passes, whereas large applications, such as the compiler itself, requires approximately 30 passes to reach a fixed point.

Second step. Reduction and optimization of the live code. Using the dynamic type information gathered during step 1, we replace dynamic dispatch with branching code to make new type flow information available. Then, each pass reconsiders all the live code in an attempt to remove the code that is no longer reachable or attempting to inline the reachable code. At the end of each pass, the type set of each expression is increasingly more accurate by type-set reduction until reaching a fixed point. The closer we get to the end of the process, the more precise our type analysis is. A fixed point is reached when no further inlining or transforming is possible. For large-size applications such as the compiler itself, the number of passes varies between approximately 10 and 25.

Third step. Final target code generation. During this last step, the field layouts into data structures representing objects are optimized. Objects that are not involved in dynamic dispatches are not equipped with the type id field. Only the used fields are generated and ordered to reduce object size. Even after these steps, some dynamic dispatch may remain, so dynamic type ids are also selected during this final step to compact switch dispatch tables.

1.4. Article overview and major contributions of the article

Section 2 details how the reachable code is gathered during the first step of our compilation strategy. Section 2.1 starts with the rather classical transitive closure of the call graph with partial evaluation to collect a large superset of the live code. Section 2.2 presents our results for type analysis of variables. Section 2.3 presents our technique to predict types inside arrays, as well as its major impact on garbage collection inside arrays.

DOI: 10.1002/spe

The second step of our strategy is detailed in Section 3. Its major point is to take advantage of CWA to get rid of virtual function tables (VFTs), shown in Section 3.1, using branching code as a replacement. Section 3.2 is dedicated to method customization and presents the choices we made to avoid code explosion, thanks to the Argument Type Set (ATS) customization of Lisaac. The measurements we present demonstrate the scalability of our approach. Then, our inlining strategy inside dispatch branching code is presented in Section 3.3 along with its two major results: the perfect translation of `ifTrue:ifFalse:` (3.3.1) and the perfect translation of `whileTrue:` (3.3.2). The more traditional loop invariant detection is developed in Section 3.4, followed by reference comparison in Section 3.5. Finally, the branch merging transformation is presented in Section 3.6.

Section 4 describes the third and final compilation step. The field order is selected in Section 4.1 to reduce the size of objects. The effective global dispatch map is used to select type ids to optimize switch branching code in Section 4.2. Section 5 is dedicated to benchmarking. The bootstrap of the SmartEiffel compiler, Section 5.1, highlights the impact of dispatch branching code without type flow analysis on a large application. Then, in Section 5.2, another large application, the bootstrap of the Lisaac compiler, is presented this time with type flow analysis. Our experiments focus on late binding in Section 5.3, cascading message sends on the same receiver in Section 5.4, calls on the `self` variable in Section 5.5, and method calls involving multiple inheritance in Section 5.6. The impact of inlining is studied in Section 5.7. Then, the last benchmark presented in 5.8 is a real MPEG2 production decoder. A handwritten C version is compared with a systematic translation into Lisaac code. Most of the related works are covered throughout the article, and Section 6 ends by presenting them. A description of our future work is given in Section 7 and Section 8 concludes.

2. FIRST STEP: GATHERING LIVE CODE

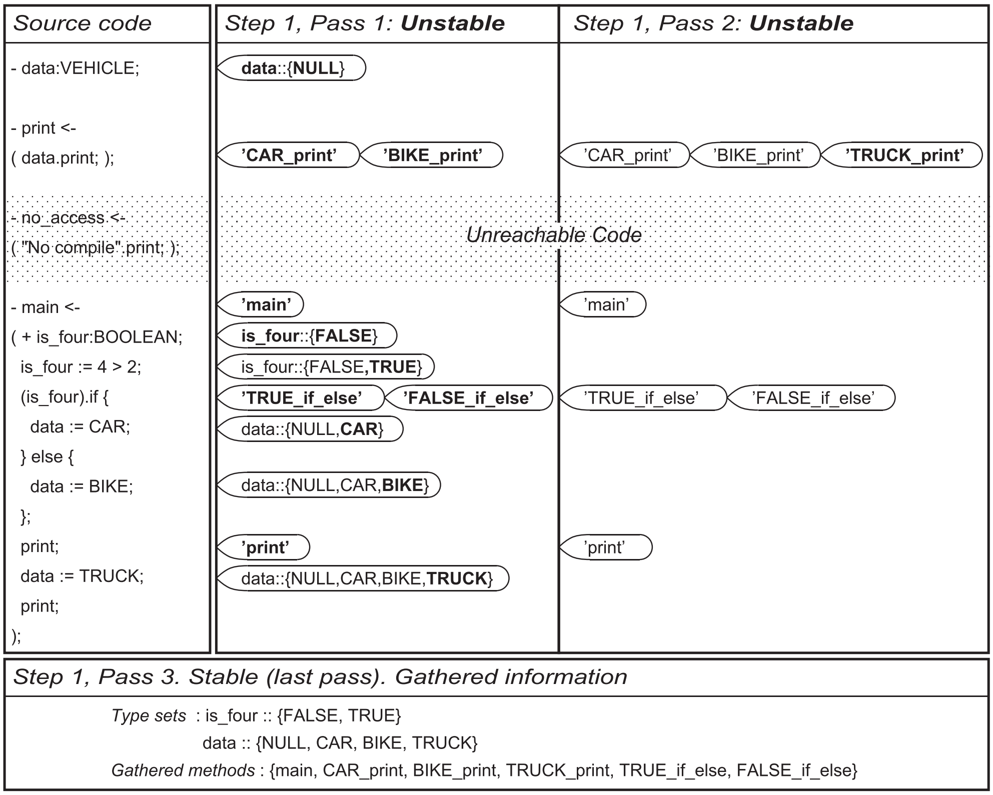
The first step in our strategy is essentially a partial evaluation [16, 17] of the program starting from the entry point. The goal is to gather all the reachable code, even a rough superset of the reachable code, from the entire source code necessary to run the application. At the same time, all expressions are tagged with a set of possible dynamic types to follow method calls by dynamic dispatch simulation.

2.1. Transitive closure of the call graph

Using the `main` function code as the starting point, we first develop and analyze the call graph of the complete source code and then compute the set of all possible dynamic types that can occur at runtime. Let us take into consideration the simplistic CAR/TRUCK example given in Figure 2 where the entry point of the graph is the main method. Any unreachable code is simply ignored and thus never compiled, avoiding the cost of unnecessary compilation. For instance, the `no_access` method in Figure 2 is never gathered. The reachable methods are stored and, each time a new possible dynamic type for the receiver is encountered, customized accordingly.

Genericity as well as multiple inheritance are processed during the first step. Genericity is treated by code duplication followed by code customization, simply taking into account effective generic parameters. Thanks to CWA, it is easy to know each possible generic derivation as well as all the possible dynamic types. Each time a method call site is visited, the lookup mechanism is simulated according to inheritance rules. For all possible dynamic types of the receiver, the corresponding method is reached and may be, in case of a new one, customized and collected. Finally, the overhead of multiple inheritance only impacts compilation time (impact of CWA on multiple inheritance is detailed in [18]).

Computation of the transitive closure of the call graph requires several passes until it reaches a fixed point. Each time a pass encounters new code or when a pass adds a new possible dynamic type for an expression, yet another pass will have to be performed. A fixed point is reached when a pass adds neither new code nor new dynamic type. This gives us a superset of the reachable code as



```
- data:VEHICLE;

- print <-
( data.print; );

- no_access <-
( "No compile".print; );

- main <-
( + is_four:BOOLEAN;
  is_four := 4 > 2;
  (is_four).if {
    data := CAR;
  } else {
    data := BIKE;
  };
  print;
  data := TRUCK;
  print;
);
```

Figure 2. Step 1: Gathering live code on the CAR/TRUCK example. The bold typeset indicates new gathered information, new source code, or some new possible dynamic types for some expressions. Notice that the default value for reference variables is NULL and FALSE for variable of type boolean.

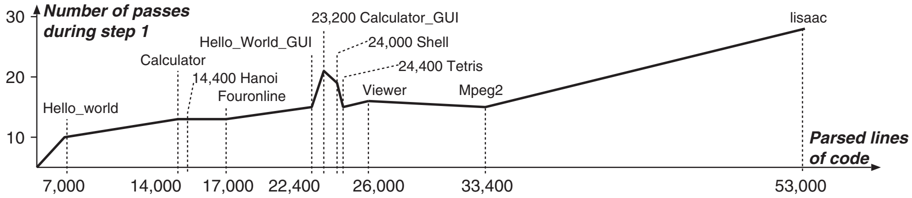


Figure 3. Number of passes before reaching a fixed point for some benchmarks.

well as a superset of the possible dynamic types for each expression. For the example in Figure 2, there are three passes, and when a fixed point is reached, the set of possible dynamic types for `data` is {NULL, CAR, BIKE, TRUCK}. As a consequence of the method call `data.print`, methods CAR_print, BIKE_print, and TRUCK_print are reachable and, consequently, added in the set of gathered methods. Note that there is no data flow analysis during step 1: data flow analysis would have removed BIKE and NULL from the set of `data`. During our first compilation step, the goal is to gather, as quickly as possible, a rough superset of the live code. Thanks to the data flow analysis of step 2, BIKE and NULL will then be removed from that set. Still in Figure 2, the dynamic types for the `is_four` boolean local variable is {TRUE, FALSE}. As the control flow statements are defined in the library as in Smalltalk or Self, methods TRUE_if_else and FALSE_if_else are gathered too, as any other ordinary methods. Our measurements presented in Figure 3 indicate that during step 1, the number of passes vary in a logarithmic manner as a function of the code size. On the entire code of our Lisaac compiler, consisting of 53,000 lines of code, we have 29 dependency passes before reaching a fixed point.

DOI: 10.1002/spe

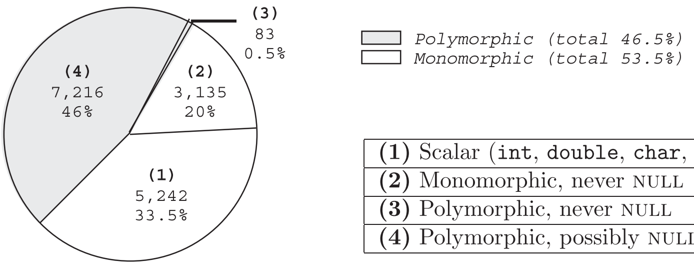
2.2. *Data type analysis for variables*

The SmartEiffel compiler carries out its type inference without type flow analysis, using a Rapid Type Analysis (RTA) algorithm [19]. Type analysis is based on the inheritance hierarchy directly available in the source code. It gives, for each static type, a superset of all the possible dynamic types. During step 1, the Lisaac compiler uses the same information to initialize its data; then, to have a better dynamic type analysis, the Lisaac compiler adds extra type flow information. The list of possible dynamic types is no longer computed for each static type but distinctively for each variable introduced in the live code, namely instance variables, formal parameters, and local and global variables. For each variable, the list of assignments to that variable is recorded and internally represented as a directed graph where variables are the nodes and assignments are the arcs. Possible leaf vertices of the directed graph are constant values, explicit object identifiers, objects creations, and the NULL value. A method call is a labeled vertex giving access to the subgraph of this method. The transitive closure of the directed graph of assignments gives all the possible dynamic types for each attribute, each formal argument, and each local variable and, consequently, all the possible dynamic types for each expression. To summarize, during step 1, we are using RTA as a flow-insensitive interprocedural type analysis for all live methods. Flow-sensitive analysis only occurs during step 2. Both type flow analysis can be related to Control-Flow Analysis (CFA) [20, 21].

As for constant values, the NULL value, denoting the absence of object, is a leaf vertex of the directed graph of assignments. Finally, each expression can be classified into three categories: either an expression can sometimes be NULL, can never be NULL, or is always NULL. This information is useful to know if an expression may cause a *call on null* error. Measurements performed on the complete source code of the Lisaac compiler shown in Figure 4 indicate that 53.5% (i.e. 33.5% + 20%) of variables are monomorphic variables. All other variables, 46.5% (i.e. 46% + 0.5%), are polymorphic. Thus, only 46.5% of variables require dynamic dispatch if they are used as the target of certain method calls. The accuracy of type analysis is essential in reducing the execution time overhead of dynamic dispatch. As shown in Section 5.3, most method call sites are usually statically resolved. From the language design point of view, it is easier to predict types when the language is statically typed and when the initialization policy of the language does not leave place for uncertainty. For example, in Lisaac and SmartEiffel, noninitialized data gets a default value. Another good language design decision is with Java and C#, whereas the programmer must initialize all local variables and all instance variables have default values, avoiding any uninitialized piece of memory.

2.3. *Data type analysis and garbage collection optimization of arrays*

To perform global type flow analysis, array read–write operations rely on the built-in NATIVE_ARRAY [E] abstract data type. The NATIVE_ARRAY [E] abstract data type was first introduced in SmartEiffel to avoid uninitialized cells and to optimize garbage collection. This abstract data type is a kind of array list: some `capacity` is given as an argument of the constructor, and the filling up is made progressively, cell by cell from left to right (see Figure 5). For type flow analysis, Lisaac considers the whole used part as a single cell. Actually, we adapted the *array smashing* method of [22] for type flow analysis. To summarize, as soon as one cell is possibly assigned with an object of type A, all cells of the used part are considered as potential holders of objects of type A.



(1) Scalar (`int`, `double`, `char`, etc.)	5,242
(2) Monomorphic, never NULL	3,135
(3) Polymorphic, never NULL	83
(4) Polymorphic, possibly NULL	7,216

Figure 4. Data type analysis for the variables of the Lisaac compiler source code after step 1.

DOI: 10.1002/spe

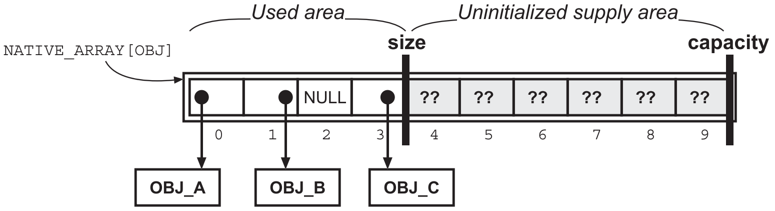
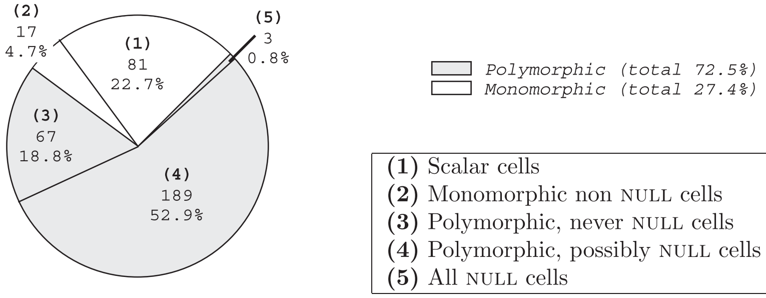


Figure 5. Arrays are filled progressively from left to right to avoid uninitialized values.



(1) Scalar cells	81
(2) Monomorphic non NULL cells	17
(3) Polymorphic, never NULL cells	67
(4) Polymorphic, possibly NULL cells	189
(5) All NULL cells	3

Figure 6. Distribution of arrays in the Lisaac compiler after step 1. Only 27.4% of arrays are monomorphic.

Although the information collected for NATIVE_ARRAYs lacks index sensitivity, it allows us to conclude whether an element can have a NULL value or not. Because NULL is considered as a particular type, the absence of the dynamic NULL type inside an array is a significant piece of information. Any method call applied on an element of this array is statically guaranteed to not be NULL; therefore, there will be no *call on null* error. Measurements performed on the Lisaac compiler, shown in Figure 6, indicate that 27.4% of arrays are monomorphic. This result is not as good as the one we obtained for variables (Figure 4), but this is not surprising as most arrays are stored in *long-life* variables (i.e. attributes or globals), which are accessible from many locations. Still in Figure 6, arrays containing only NULL values are empty hash maps.

Most garbage collector algorithms [23] need to scan through all the accessible objects, like the *copying collector* or the *mark-and-sweep* garbage collectors. For instance, the *mark-and-sweep* collector must walk through all accessible arrays during the mark phase. As a result of our filling-up strategy, the supply memory area of arrays (Figure 5) is unreachable. Our GC (SmartEiffel and Lisaac) uses this knowledge to avoid scanning of that area. This makes the GC faster and prevents it from accidentally marking inaccessible objects. Furthermore, the type flow information of the array elements can be integrated to the GC as it is already the case for the objects' attributes [24]. SmartEiffel generates a specialized and precise marking function for every object type and excludes interpretation during execution. We equipped the source code of the SmartEiffel garbage collector to examine the impact of arrays on memory footprint. To have an indicative execution, we are using the entire source code of the SmartEiffel compiler itself, which is 180,000 lines of Eiffel source code during its own bootstrap. The self recompilation of the compiler is a very good benchmark because it uses many arrays and requires approximately 330 Mb of memory during the process. Furthermore, the garbage collector is triggered 32 times while compiling the compiler. For the following measurements, only arrays of references are considered, because other arrays, for example arrays of integers or arrays of characters, are not directly concerned by our type flow analysis technique. Additionally, the SmartEiffel garbage collector does not even scan the content of arrays of scalars. We modified the marking procedure for the content of arrays to count the number of marked arrays during one recompilation. The measurement shows that the GC processes 6,399,198 arrays. The total size of the corresponding used area scanned is of 12,548,963 cells. As the total `capacity` of processed arrays is of 21,714,957 cells and because the supply area is not scanned, the GC avoids scanning 9,165,994 cells, which signifies a gain of 42% [25].

DOI: 10.1002/spe

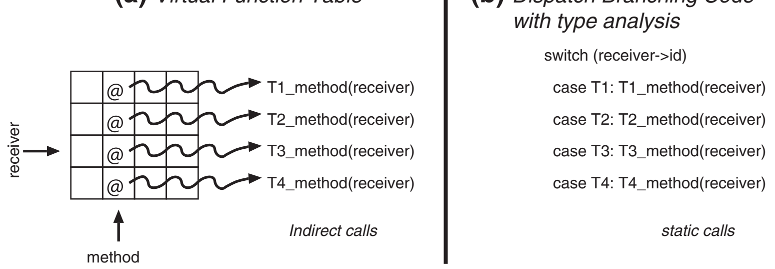
3. SECOND STEP: OPTIMIZING THE CLOSED WORLD

Because of the results of the previous step, we can begin working in a *closed world*: the internal representation of the source code we are working on is a superset of the reachable code. Each expression has a finite set of possible dynamic types including the information regarding the NULL value. At this time, the gathered information is pessimistic, and the goal of the second step is to refine it. During the second compilation step, we carry out a number of optimizations, among which we will mention: those who have an important impact on the runtime and those related to object-oriented languages. Special treatment of operating system dedicated features that are specific to Lisaac is beyond the scope of this article. Only general purpose optimizations are described here.

3.1. Dispatch branching code and general survey of step 2

Needless to say that the implementation of late binding is crucial for the performance of object-oriented languages. The most popular implementation relying on a VFT [26, 27] is often used for languages such as Java, C#, or C++. Figure 7(a) illustrates the necessary indirect function call of the VFT implementation. Each value of the table is a function pointer leading to indirect calls. Figure 7(b) illustrates the dispatch branching code previously introduced in SmartEiffel [3]. This implementation, which is only possible under CWA, allows inlining or transforming of static calls inside each case branch. As we obtained exceptional results with the SmartEiffel compiler, we decided that for the Lisaac compiler to experiment with a similar strategy, using more inlining and code customization, and to add data flow information. Internally, the abstract representation of dynamic dispatch is a *switch-like* representation similar to the one presented in Figure 7(b). The most significant reason for this choice is the possibility to perform inlining inside case branches. As the same type of internal switch statement is also used for any written switch statement of the users source code, the type flow analysis is performed indifferently. The type set information for each expression gathered during step 1 is used to build the dispatch branching code. This internal representation of the code constitutes the initial state of step 2. Step 2 is composed of passes on the entire code, to inline or to transform the code, also using data flow analysis, making as many passes as necessary to reach a fixed point. A fixed point is reached when a complete traversal of the code is performed without any transformation.

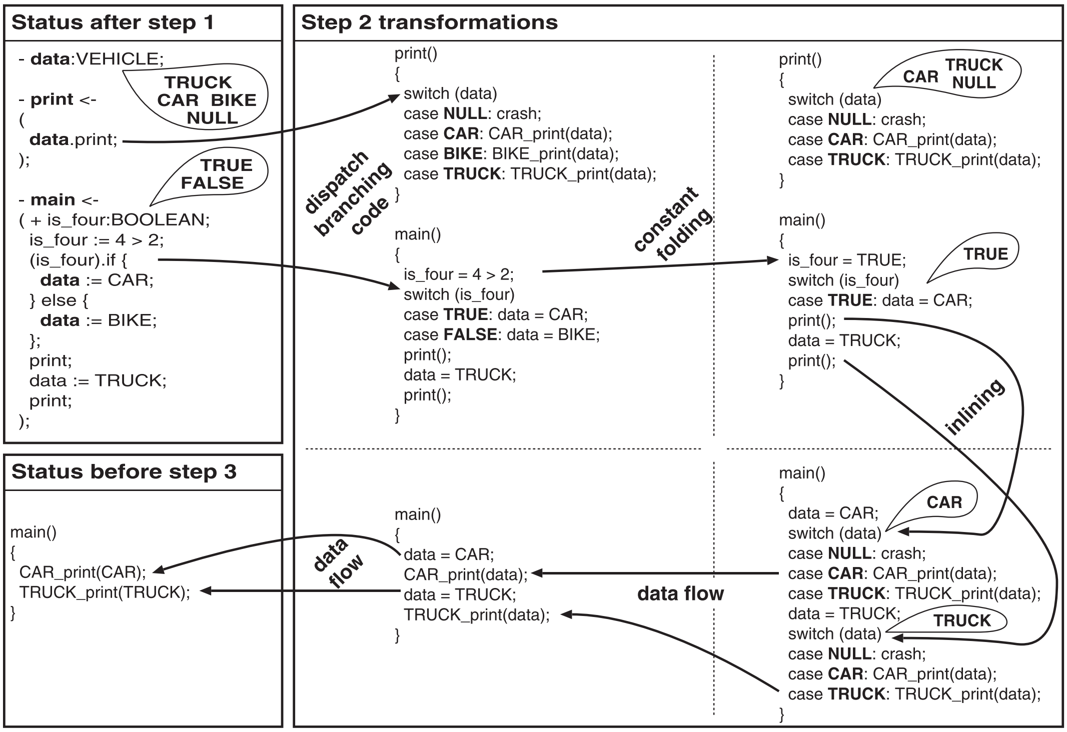
The most common code transformation occurs inside the case branches of the dispatch branching code. Indeed, in such a case branch, the receiver's dynamic type is resolved, and the corresponding static call is a candidate for inlining or transforming. Simple data flow techniques also allow code simplification as shown in Figure 8, with the previously used CAR/TRUCK example. Now that we have given an overview of step 2, the next subsection presents the method customization issue before a detailed presentation of the inlining strategy (Section 3.3), loop invariant detection (Section 3.4), reference comparisons (Section 3.5), and dispatch branch merging (Section 3.6). All the transformations performed during step 2 allow us to better specialize the code and, when possible, to reduce the dynamic type sets. For instance, the dynamic type set of data in Figure 8 shrinks progressively during the passes of step 2. Whereas the type sets are only growing during passes of step 1, the opposite happens during passes of step 2; the type sets become smaller and smaller.



```
switch (receiver->id)
  case T1: T1_method(receiver)
  case T2: T2_method(receiver)
  case T3: T3_method(receiver)
  case T4: T4_method(receiver)
```

Figure 7. Virtual function table versus dispatch branching code with type analysis.

DOI: 10.1002/spe



Status after step 1:

```
- data:VEHICLE;
- print <-
(
  data.print;
);
- main <-
( + is_four:BOOLEAN;
  is_four := 4 > 2;
  (is_four).if {
    data := CAR;
  } else {
    data := BIKE;
  };
  print;
  data := TRUCK;
  print;
);
```

Step 2 transformations:

```
print()
{
 switch (data)
 case NULL: crash;
 case CAR: CAR_print(data);
 case BIKE: BIKE_print(data);
 case TRUCK: TRUCK_print(data);
}
main()
{
 is_four = 4 > 2;
 switch (is_four)
 case TRUE: data = CAR;
 case FALSE: data = BIKE;
 print();
 data = TRUCK;
 print();
}
```

```
print()
{
 switch (data)
 case NULL: crash;
 case CAR: CAR_print(data);
 case TRUCK: TRUCK_print(data);
}

main()
{
 is_four = TRUE;
 switch (is_four)
 case TRUE: data = CAR;
 print();
 data = TRUCK;
 print();
}
```

```
main()
{
 data = CAR;
 switch (data)
 case NULL: crash;
 case CAR: CAR_print(data);
 case TRUCK: TRUCK_print(data);
 data = TRUCK;
 switch (data)
 case NULL: crash;
 case CAR: CAR_print(data);
 case TRUCK: TRUCK_print(data);
}
```

```
main()
{
 data = CAR;
 CAR_print(data);
 data = TRUCK;
 TRUCK_print(data);
}
```

Status before step 3:

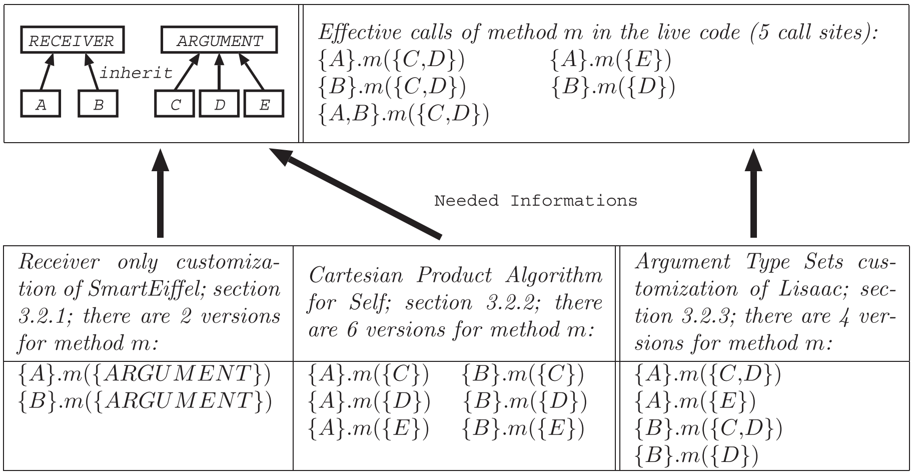
```
main()
{
 CAR_print(CAR);
 TRUCK_print(TRUCK);
}
```

Figure 8. The CAR/TRUCK example during step 2 (constant folding, inlining and data flow).

3.2. *Method customization*

Method customization consist in the definition of a new version of a method, adapted to the characteristics of the calling site. Thereafter, the customized method is only used for calls with the same characteristics as the original calling site. Methods are customized in a way that impacts the size of the generated code as well as the level of specialization of that code. The more the code is specialized, the more it can be transformed efficiently. On the other hand, too much specialization makes the code so large that one cannot expect to translate very large applications. Let us first review the SmartEiffel strategy.

3.2.1. Customization according to the receiver only (SmartEiffel). Customization according to the type of the *receiver only* consists in the definition of one customized method for each possible dynamic type of the receiver [3, 19, 28]. No possible variation of the arguments type intervenes. Hence, for one given method that can be called polymorphically, the number of actually defined customized methods is equal to the number of possible dynamic types for the receiver. The body of each customized method is specialized for only one dynamic type, making each call on the `self` variable direct static calls. Inside the body, when the method called is small enough, it can be inlined or, in the best case, statically computed. A typical example is the static computation of a boolean expression, which makes one branch of an *if_then_else* statement unreachable. This results in additional dead code elimination and may impact call sites outside of that method body. In such a case, a new traversal of the entire code is performed to take into account possible simplifications: receiver type-set reduction or extra inlinings. For that reason, the specialization of method bodies is performed as early as possible, not only during final code generation. As shown in the example of Figure 9, two different methods are created using the inheritance tree. One version of m is called when the dynamic type of the receiver is A and the other when the dynamic type of the receiver is B. Each version of m can be called with all possible types for the argument (i.e. the static $ARGUMENT$ type). The SmartEiffel strategy, although perfectly scalable, lacks specialization on arguments to take advantage of the gathered information.



Effective calls of method m in the live code (5 call sites):
$\{A\}.m(\{C,D\})$ $\{A\}.m(\{E\})$
$\{B\}.m(\{C,D\})$ $\{B\}.m(\{D\})$
$\{A,B\}.m(\{C,D\})$

Receiver only customization of SmartEiffel; section 3.2.1; there are 2 versions for method m:	*Cartesian Product Algorithm for Self; section 3.2.2; there are 6 versions for method m:*	*Argument Type Sets customization of Lisaac; section 3.2.3; there are 4 versions for method m:*
$\{A\}.m(\{ARGUMENT\})$ $\{B\}.m(\{ARGUMENT\})$	$\{A\}.m(\{C\})$ $\{B\}.m(\{C\})$ $\{A\}.m(\{D\})$ $\{B\}.m(\{D\})$ $\{A\}.m(\{E\})$ $\{B\}.m(\{E\})$	$\{A\}.m(\{C,D\})$ $\{A\}.m(\{E\})$ $\{B\}.m(\{C,D\})$ $\{B\}.m(\{D\})$

Figure 9. Example to compare the customization of some method *m* with one argument.

3.2.2. Customization with Cartesian Product Algorithm (Self/Agesen). The Cartesian Product Algorithm (CPA) for the Self language [7, 29] customizes methods for all possible types of the receiver, but also for all possible types of the arguments. This results in a better specialization of method bodies because calls on formal parameters are also specialized. Unfortunately, more customized methods need to be developed. Invocation of methods is also made more difficult because the dynamic types of arguments are involved in the dispatch mechanism together with the dynamic type of the receiver. As shown in Figure 9, the receiver type set is $\{A, B\}$, and the set of the possible argument types is $\{C, D, E\}$. Therefore, there are 2×3 versions of method *m* that are created. With CPA, each customized method is highly specialized. Before calling a method, one must dispatch not only with the type of target but also with the types of arguments. The CPA strategy is a flexible and dynamic approach designed under the OWA. The main drawback of this approach is that it creates too many customizations. In the example given in Figure 9, the method customization $\{B\}.m(\{E\})$ is not necessary under CWA.

3.2.3. Argument Type Set customization (Lisaac). Argument Type Set customization of Lisaac is a trade-off between receiver only customization of SmartEiffel (3.2.1) and generalized CPA customization of Self (3.2.2). For a given call site, ATS uses the type set of the receiver as well as the type set of each effective argument. The number of customized methods that are possibly called matches the number of possible types of the target at that call site. All customized methods of that call site have the same signature for arguments. Actually, each argument is tagged with the possible type set for that argument at that call site. By doing so, only the dynamic type of the receiver is involved in the dispatch mechanism. The type information gathered for arguments at a given call site is enclosed in the bodies of the corresponding customized methods. A pool of customized methods is updated throughout the compilation process. As soon as a simplification occurs to a method body, a complete traversal of the code is necessary to take into account possible modifications in other methods. A customized method is shared by two call sites only if the type sets for all effective arguments are identical.

3.3. Transformation inside dynamic branching code

By its very nature, VFT usage implies making indirect calls to true functions, even for a simple field access. To permit inlining of simple operations, we are obliged to use either an *if_then_else* balance tree or a *switch* statement. For polymorphic calls with only two possibilities, Lisaac uses a simple *if_then_else* (see 3.3.1). When there are more possibilities, Lisaac generates a *switch* statement to allow a constant time selection (see 4.2).

DOI: 10.1002/spe

A block closure is an anonymous function or an anonymous procedure that is saved along with the current bindings from enclosing blocks for later invocation. When it cannot be statically resolved, a block closure is typically implemented by saving both the function and any activation records that contain variables referenced by the function. The closure creates additional implicit references to the bindings closed over and hence must be accounted for in any memory management scheme (i.e. closures are as costly as true objects). The closure itself is an object that must be managed and may have either a dynamic extent or an indefinite extent depending on whether it is only used by inner blocks of the creating block or passed out of the creating block. Because dynamic closure management is very costly both in terms of memory and execution time for calling the delayed code [30], we have designed our inlining strategy to maximize static resolution of block closures.

Our inlining algorithm features six possibilities according to the encountered call:

Case #1: *The call is a tail-recursive call.* In such a situation, the call is favorably replaced with a `goto` statement at the beginning of the enclosing method. By doing so, the enclosing function is no longer recursive and is thus a possible candidate for subsequent situations. Note that it is not possible, in the general case, to remove tail recursion when using VFTs.

Case #2: *The called method is recursive.* The static call to the corresponding method is left unchanged. The rationale is that the stack is convenient to implement the recursion. Although inlining may be possible under unusual conditions, we prefer to not inline here. When a lexical closure is part of the call, we have to manage runtime structures to capture the context. Usually, this situation is quite rare, and the catch is more commonly reduced to the current receiver.

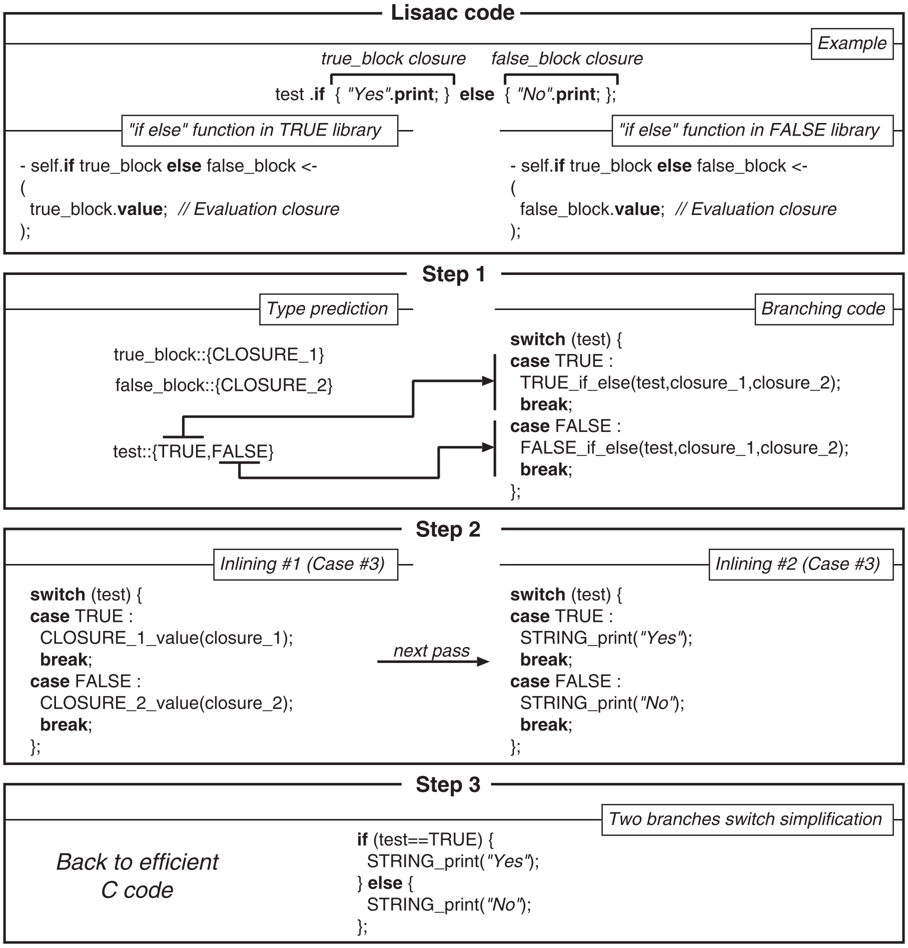
Case #3: *A lexical closure is used in receiver and/or some arguments.* A lexical closure may for example use a local variable of the enclosing context. Inlining makes available the usage of that closure into the current context. Applying transitive transformations without encountering the previous case #2 allows a static implementation of closures.

Case #4: *The method call is unique.* Thanks to CWA, case #4 is straightforward and avoids the unnecessary function definition with the unique associated method call.

Case #5: *The method called is small enough.* A compiler option provides the ability to select the right balance between either performance or binary code size. The default value is the result of the experimentation presented in Section 5.7.

Case #6: *All other cases.* In all other situations, the call is a direct static call to the function.

3.3.1. Transformation of the if_then_else conditional. As in Smalltalk or Self, the Lisaac *if_then_else* statement is defined in the library, not in the actual language. In the BOOLEAN class, this conditional statement is actually an abstract method consisting of two arguments, both of closure type. The first argument represents the *then* part and the second argument the *else* part. There are two definitions of this method, one in the TRUE class and the other in the FALSE class. The definition of the TRUE class only executes the *then* closure, whereas the definition of the FALSE class only executes the *else* closure. For efficiency reasons, Smalltalk and Self compilers use a special treatment with hard-coded primitives to handle similar *if_then_else* conditional statements. As a result of our compilation strategy without VFTs and the transformation rules we selected, it is not necessary to handle conditional statements of this type with a special case inside the compiler. We are thus able to obtain the best possible translation without using any compiler tricks, by simply applying our general compilation scheme. Figure 10 details all steps of the compilation process for an *if_then_else* statement. Dynamic dispatch on the boolean value is first translated with a two branch `switch`, one for TRUE and the other one for FALSE. The last step replaces the `switch` with a hard-coded `if` statement. As a result, the generated code is as efficient as handwritten C code. Notice that the transformation of a two branch `switch` into a simple conditional statement occurs, generally not only for booleans but for all two element type sets dispatch call sites.



```
test .if { "Yes".print; } else { "No".print; };
```

```
- self.if true_block else false_block <-
(
  true_block.value;  // Evaluation closure
);
```

```
- self.if true_block else false_block <-
(
  false_block.value;  // Evaluation closure
);
```

```
true_block::{CLOSURE_1}
false_block::{CLOSURE_2}
test::{TRUE,FALSE}
```

```
switch (test) {
case TRUE :
  TRUE_if_else(test,closure_1,closure_2);
  break;
case FALSE :
  FALSE_if_else(test,closure_1,closure_2);
  break;
};
```

```
switch (test) {
case TRUE :
  CLOSURE_1_value(closure_1);
  break;
case FALSE :
  CLOSURE_2_value(closure_2);
  break;
};
```

```
switch (test) {
case TRUE :
  STRING_print("Yes");
  break;
case FALSE :
  STRING_print("No");
  break;
};
```

```
if (test==TRUE) {
  STRING_print("Yes");
} else {
  STRING_print("No");
};
```

Figure 10. Steps to compile the library-defined *if_then_else* conditional into the best possible C code.

3.3.2. Transformation of the while_do loop. As for conditional statements, all loop statements are defined in the library and are not part of the actual language. Once more, applying our general compilation strategy allows us to obtain the best translation, even for loop statements. As an example, Figure 11 details of the steps necessary to translate the *while_do* statement. During the cascade of transformations, applying case #1, for the tail-recursive call, leads to the `goto` loop. The apparent final step reaches the corresponding `while` statement. Again, the translation to handwritten C code is reached.

3.4. Loop invariant detection

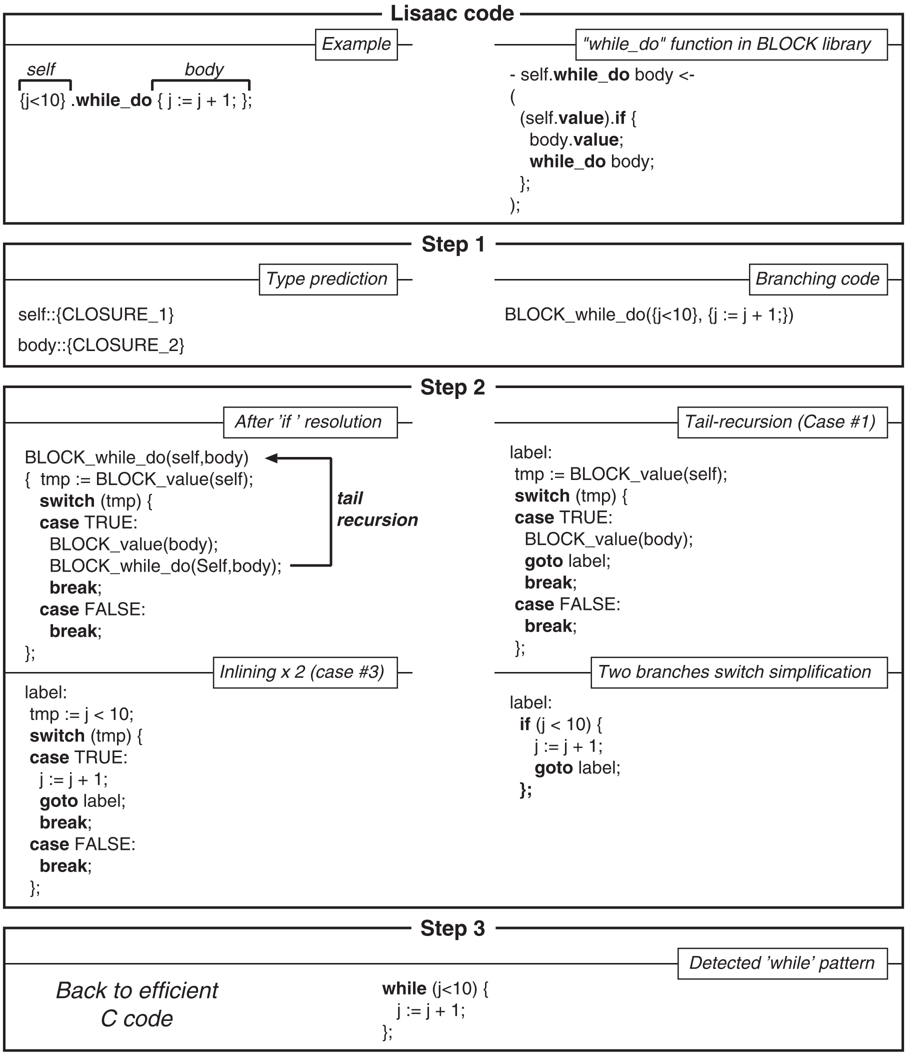
Loop invariant detection is better achieved when the body of the loop is fully defined as it is the case under CWA. For example, let us consider the following C++ like piece of code:

```
STRING string; // Declaration of the string attribute
void method() {
 int j;
 this->string = new STRING();
 this->string->length = 5;
 this->string->storage = "Hello";
 j = 0;
 while (j < this->string->length) {
  io->put_char(this->string->storage[j]);
  j ++;
 };
}
```

DOI: 10.1002/spe

In the previous code, `string` is the attribute of the `this` object. In a C++ like language, the `put_char()` method can have access to the `this` pointer and thus is able to modify the `length` and/or the `storage` attribute. Under CWA, the compiler is aware of the `put_char()` method invoked on the `io` object. If `put_char()` does not modify the attributes of `string`, the previous `method` code can be optimized as follows, using constant propagation and loop invariant extraction:

```
void method() {
 int j;
 this->string = new STRING();
 this->string->length = 5;
 this->string->storage = "Hello";
 j = 0;
 { char *tmp=this->string->storage;
   while (j < 5) {
    io->put_char(tmp[j]);
    j ++;
   };
 };
}
```



```
{j<10} .while_do { j := j + 1; };
```

```
- self.while_do body <-
(
  (self.value).if {
    body.value;
    while_do body;
  };
);
```

```
self::{CLOSURE_1}
body::{CLOSURE_2}
```

```
BLOCK_while_do({j<10}, {j := j + 1;})
```

```
BLOCK_while_do(self,body)
{  tmp := BLOCK_value(self);
   switch (tmp) {
   case TRUE:
     BLOCK_value(body);
     BLOCK_while_do(Self,body);
     break;
   case FALSE:
     break;
};
```

```
label:
 tmp := BLOCK_value(self);
 switch (tmp) {
 case TRUE:
   BLOCK_value(body);
   goto label;
   break;
 case FALSE:
   break;
 };
```

```
label:
 tmp := j < 10;
 switch (tmp) {
 case TRUE:
   j := j + 1;
   goto label;
   break;
 case FALSE:
   break;
 };
```

```
label:
  if (j < 10) {
     j := j + 1;
     goto label;
  };
```

```
while (j<10) {
   j := j + 1;
};
```

Figure 11. Steps to compile the library-defined *while_do* loop into the best possible C code.

DOI: 10.1002/spe

Then, the five steps loop can be traditionally unrolled into the following:

```
void method() {
 this->string = new STRING();
 this->string->length = 5;
 this->string->storage = "Hello";
 io->put_char('H');
 io->put_char('e');
 io->put_char('l');
 io->put_char('l');
 io->put_char('o');
}
```

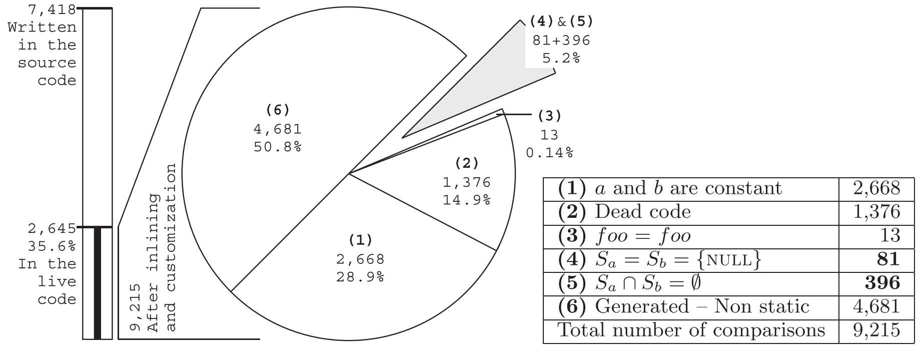
3.5. *Optimization of reference comparisons*

Method customization obviously implies more code as it creates new *specialized* code. For instance, within any method, the receiver (i.e. `self` or `this`) cannot be NULL and has exactly one determined dynamic type. When a polymorphic method call site is broken up with the corresponding dispatch branching code, the non-NULL target expression also gets one unique possible dynamic type within each `case` branch of the `switch`. Remarkably, we observed that many reference comparisons became constant, that is, either always true or always false, leading to code removal or at least to code reduction. Indeed, it is not common for a programmer to consciously write an == expression (or a != expression), which is always true or always false. It is quite common, in object-oriented software, to write a template method pattern [31] with a comparison expression, which becomes statically computable in some subclasses. As each expression is tagged with the set of all the possible dynamic types, the NULL value being a possible element of the set, rules to decide when a comparison expression is constant or not constant are as follows. In this instance, we will take into consideration a comparison of the form `a == b` or `a != b` where S_a and S_b are the corresponding type sets. When $S_a = S_b = \{\text{NULL}\}$, the comparison expression is constant, thus `a == b` always yields true and `a != b` always yields false. The second situation making a comparison constant occurs when $S_a \cap S_b = \emptyset$. The intersection of S_a and S_b being the empty set, `a == b` always yields false and `a != b` always yields true. Under all other circumstances, the comparison cannot be statically determined. Measurements on the Lisaac compiler are presented in Figure 12. During the parsing of the whole source code of the Lisaac compiler, 7418 comparisons are encountered, but only 2645 of those written comparisons are reachable. The latter, as a result of inheritance and code customization, are then transformed into 9215 comparisons. Still in Figure 12, the comparison categories from `(1)` to `(5)` disappear. Category `(1)` represents comparisons of integer constants or character constants. Categories `(4)` and `(5)` are the direct consequence of type flow analysis as previously explained. Category `(3)` is for similar access to the same data, the same variable, or the same field access on both sides of the comparison operator. As most comparisons are involved in *if_then_else* statements, detecting a constant comparison makes it possible to remove either the *then* block or the *else* block. This type of code removal contributes to offset the code size explosion due to method customization. The comparisons of category `(2)` in Figure 12 are part of the removed dead code. As a result, 49.2% of comparisons disappear, and only 50.8%, category `(6)`, are part of the executable.

3.6. *Branch merging*

Invariant reference detection is used to merge sequential `switch` dispatchs when two or more sequential method call sites apply to the same unchanged `receiver` expression. When the dynamic type of the receiver cannot change, it is possible to merge the dispatching code itself to avoid having multiple dynamic type selections. An example of this would be as follows:

```
receiver.method_1;
receiver.method_2;
```



(1) a and b are constant	2,668
(2) Dead code	1,376
(3) $foo = foo$	13
(4) $S_a = S_b = \{\text{NULL}\}$	**81**
(5) $S_a \cap S_b = \emptyset$	**396**
(6) Generated – Non static	4,681
Total number of comparisons	9,215

Figure 12. Static evaluation of `a == b` or `a != b`.

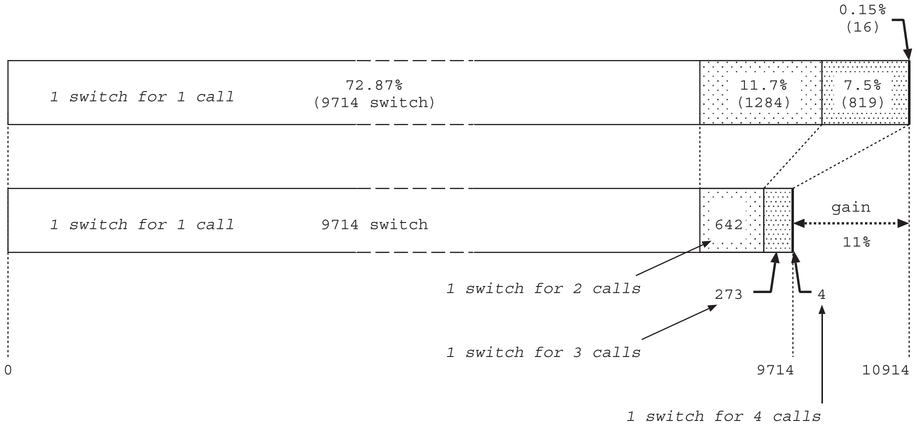


Figure 13. Branch merging distribution during bootstrap of Lisaac.

If the local type analysis for `receiver` predicts a dynamic family type set $\{A, B\}$, the following branching code on the left is merged into a single `switch`:

```
switch (receiver->id) {
case A:
    A_method_1(receiver); break;
case B:
    B_method_1(receiver); break;
};

switch (receiver->id) {
case A:
    A_method_2(receiver); break;
case B:
    B_method_2(receiver); break;
};
```

$\xrightarrow[\textit{merging}]{}$

```
switch (receiver->id) {
case A:
    A_method_1(receiver);
    A_method_2(receiver);
    break;
case B:
    B_method_1(receiver);
    B_method_2(receiver);
    break;
};
```

Actually, this merging optimization renders Smalltalk's cascading method calls notation automatic. This type of multiple method calls on the same target is quite frequent with object-oriented programming. Measurements taken on the Lisaac compiler indicates that 11% of polymorphic method calls are averted (see details of distribution in Figure 13).

4. THIRD STEP: BACK-END OPTIMIZATIONS

The final step of our compilation strategy is dedicated to back-end optimizations. To simplify this presentation, the target language is represented by the C language.

DOI: 10.1002/spe

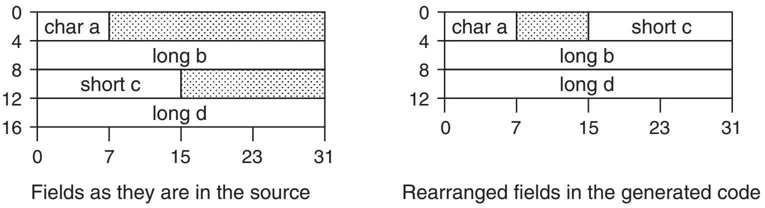


Figure 14. Rearranging a data structure can avoid many memory fragmentation.

4.1. Rearranging data

As compiler writers, we are undeniably involved in language design. A decision made during the design of the language may have drastic effects at compile time. Allowing the compiler to change the order of the fields within a structure or an object makes several optimizations possible. In contrast, such a decision is a constraint for programmers. A suitable balance between compiler writer constraints and language designer requests must be attained. As with SmartEiffel and Lisaac, the programmer cannot rely on field order to write his code. As the attributes of objects can be shifted or even removed by the compiler, it is not permitted to have a pointer pointing towards the interior of a structure. Structures or objects must always be pointed to globally. Consequently, the compiler is able to rearrange the fields of a structure to fit within the memory layout and minimize memory fragmentation. The same type of data reorganization can also be applied on local variables to save stack space. The example in Figure 14 presents a structure with four fields named `a`, `b`, `c`, and `d`. When the order of the original source code is kept unchanged the total memory size needed is 16 bytes with 5 bytes of unused memory padding. When reorganized as shown on the right side of Figure 14, the total size for entire structure is 12 bytes with only 1 byte of unused memory padding. While keeping the alignment, the rearranged structures are noticeably shorter and may fit better into a processor's cache. In a reduced data structure, the maximum number of lost bytes is equal to the size of a *processor-word - 1* and the lost bytes do not depend on the number of fields. For instance, on a 32-bit processor (4 bytes), the maximum loss is 3 bytes no matter what the number and size of fields are. On a 64-bit processor, the maximum loss of memory is 7 bytes per structure. More commonly, let 2^P be the size in bytes of a machine word ($P = 2$ for a 32-bit processor). Assuming the fields are sorted by increasing size into the structure, a memory lost occurs when the size of fields changes. Changing size from 2^i to the next size 2^{i+1} implies a maximal loss of 2^i. Changing sizes from 2^i to 2^j, with $j > i$, induces a loss of $\sum_{k=i}^{j-1} 2^k$. As a consequence, the total loss using words of size 2^p is $\sum_{k=0}^{p-1} 2^k$, that is, $2^p - 1$.

4.2. Dynamic type id selection and switch optimization

The traditional translation of `switch` statements into assembly code relies on a jump address table indexed with the switched value. Let `byte_count` be the number of byte processor's words; the pseudo assembly generated code for a `switch` statement is as follows:

```
   static void *jump_table[] = {&id_T1,&id_T2,&id_T3};

(1)     index := receiver->id - first_case_id
(2)     if ((unsigned)index > last_case_id) then
(3)      goto default_case
(4)     endif
(5)     goto (jump_table[index * byte_count])
   id_T1: T1_method(receiver); goto after_dispatch;
   id_T2: T2_method(receiver); goto after_dispatch;
   id_T3: T3_method(receiver); goto after_dispatch;
(6)  default_case:
        ...
   after_dispatch:
        ...
```

As the switched value is the integer dynamic type id, the compiler is able to select id sets to optimize the generated assembly code. Because of CWA and global analysis, all the possible switch sites are known at compile time. The dynamic type sets involved in method calls are naturally segregated into different families of objects. As an example, an application using fruit and vehicles probably will not combine those objects. As a consequence, some method calls are dispatching only fruit, whereas other method calls are dispatching only vehicles. No method call handles fruit and vehicles at the same time. It is thus possible to reuse the same ids in the fruit object family and in the vehicles object family. The id only needs to identify the object inside its family, not globally. Our numbering strategy is similar to *selector coloring* [8, 32] and is the inverse of *selector table indexing* [33, 34]. Because of the `switch` implementation, the goal of the type id numbering is first to minimize the jump table size and second to try to use values as small as possible, the best being to start at 0. When the smallest number of a family is close enough to zero, it is possible to remove the subtraction instruction *(1)* of the `switch` assembly code. The used heuristic gives the priority to the largest families, which are then sorted according to frequency of written method calls. When a family is completely separated from other families (i.e. when all the types of a family are never used in other families), the numbering can start at 0. If one same type is used in several families, its id is selected to be unique across families and minimized as much as possible. While we are implementing dynamic dispatch, all cases are associated with one dynamic type and nothing else can happen. It is not possible to go out of range of the jump table. The `default_case` is thus useless and removed (the lines *(2)*, *(3)*, *(4)*, and *(6)* are removed).

4.3. Dynamic type id selection on the compiler example

Measurements on the Lisaac compiler indicate that 315 types of objects may exist at runtime. Amongst those 315 possible dynamic types, only 120 are involved in dynamic dispatch or subtype testing (on subtype testing, see [35, 36]). Dynamic types that are not involved in dynamic dispatch or subtype testing are not numbered, and corresponding objects do not have the *id* field (object size reduced). On the compiler example, nonidentified types represent 62% of dynamic types. We have also remarked that most applications have a similar important ratio of types that are not involved in dynamic dispatch. Numbering of the 120 dynamic types involved in dynamic dispatch is presented in Figure 15. The upper section of the figure indicates that *id* 0 is used for 13 different types, that *id* 1 is used for 12 different types, and so on. The lower section of Figure 15 gives the distribution of dispatch call sites. There are 10,664 dispatched call sites or subtype tests. Each horizontal line represents a family of call sites, and a continuous line indicates that there is no break in the dispatch sequence. The largest family of call sites has 8044 members that represent 75.43% of call sites, and this family operates in the range [0–1] of *ids*. Most of them are *if_then_else* dispatch call sites (see 3.3.1). As one can discern, the result is remarkable, and there are few breaks in sequences. Also note that 46 call sites (0.43%) operate on the complete *ids* distribution.

5. EXPERIMENTS

Our compilation strategy is a combination of several ideas and techniques. Optimizations performed in a given pass may interact with other optimizations in the same or other passes [37]. As one optimization may expose opportunities for another optimization, measuring the impact of a *single* aspect of the compilation process is very difficult. For example, it is not that simple to disable type flow analysis on collections to measure the sole impact of that aspect.

5.1. Bootstrap of the SmartEiffel compiler

The bootstrap of the SmartEiffel compiler (detailed results published in [13]) is summarized in Figure 16. Each step compiles the same 50,000 lines of Eiffel source code. The first compiler on the left-hand side is the Eiffel/S 1.3 commercial compiler, which works under the open world assumption with a VFT implementation of the dynamic dispatch. The far-left compilation step is running the algorithms and data structures of Eiffel/S. The remaining compilation steps are running the algorithms and data structures of SmartEiffel. The binary code of SmartEiffel#`0` is using the

DOI: 10.1002/spe

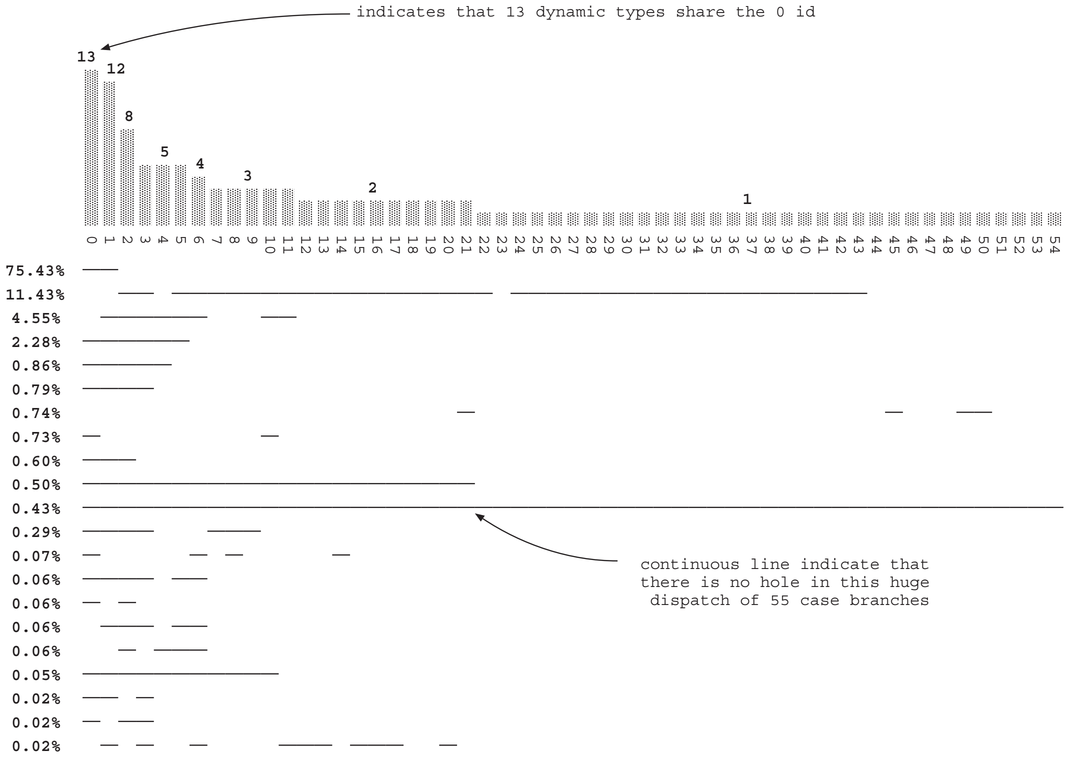


Figure 15. Numbering of dynamic types involved in dynamic dispatch on the Lisaac compiler example.

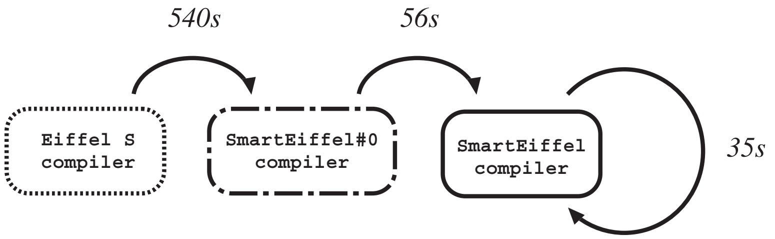


Figure 16. The bootstrap of SmartEiffel indicates a speed factor gain of 1.6 ($56s/35s = 1.6$).

VFTs that are coming from the Eiffel/S translation. SmartEiffel#0 is also using the Eiffel/S runtime libraries for memory allocation as well as the Eiffel/S objects layout. The compilation step from SmartEiffel#0 to the first stabilized SmartEiffel compiler, the step where the fixed point is reached, is the best place to perform the comparison. Because SmartEiffel does not perform type flow analysis, the 1.6 speed gain factor of Figure 16 is the gain obtained with RTA [19] and dynamic branching code [3, 13].

5.2. Bootstrap of the Lisaac compiler

The previous versions of the Lisaac compiler were written in SmartEiffel. To bootstrap, we translated the source code of the compiler into Lisaac code. The Lisaac compiler is a considerably large application of approximately 53,000 lines of Lisaac source code. We compare in Figure 17 the latest version of the Lisaac compiler written in SmartEiffel (Lisaac^{SE}) with the first bootstrapped version of the Lisaac compiler (Lisaac^{LI}). Because the translation was systematically performed, the comparison applies on the same algorithms with similar structures. The Lisaac^{SE} binary is produced by the SmartEiffel compiler, which does not feature type flow analysis whereas Lisaac^{SE} does. As a result, the number of monomorphic method calls has improved from 91% to 98.3%. As shown in Figure 17, the percentage of extra monomorphic method calls represents polymorphic method calls

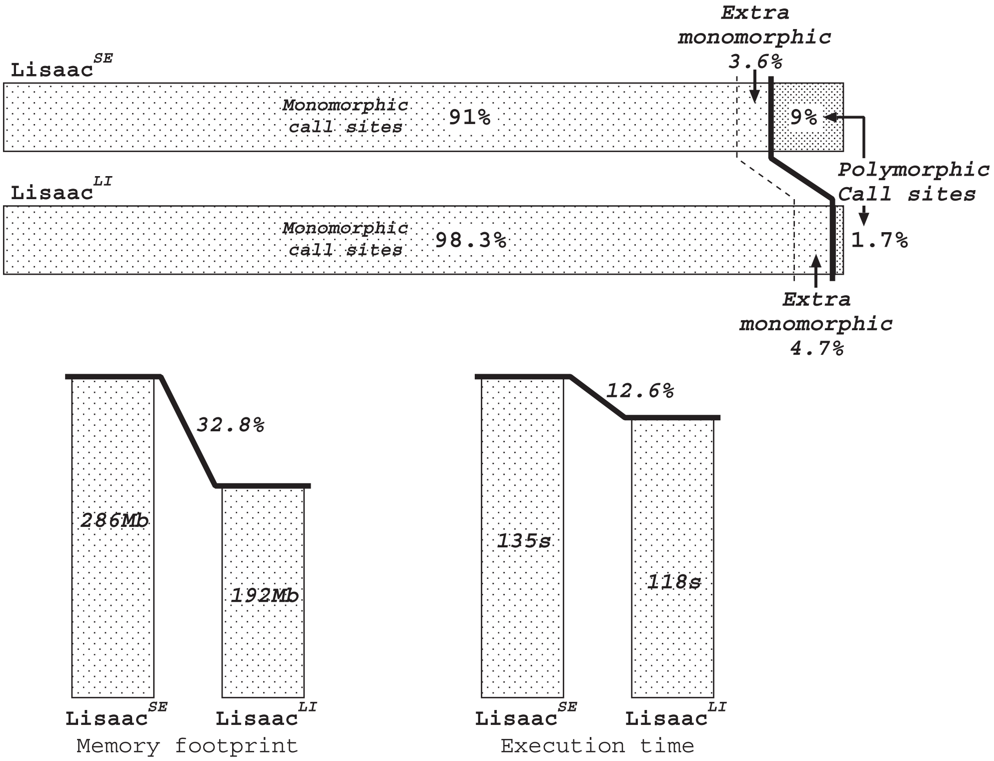
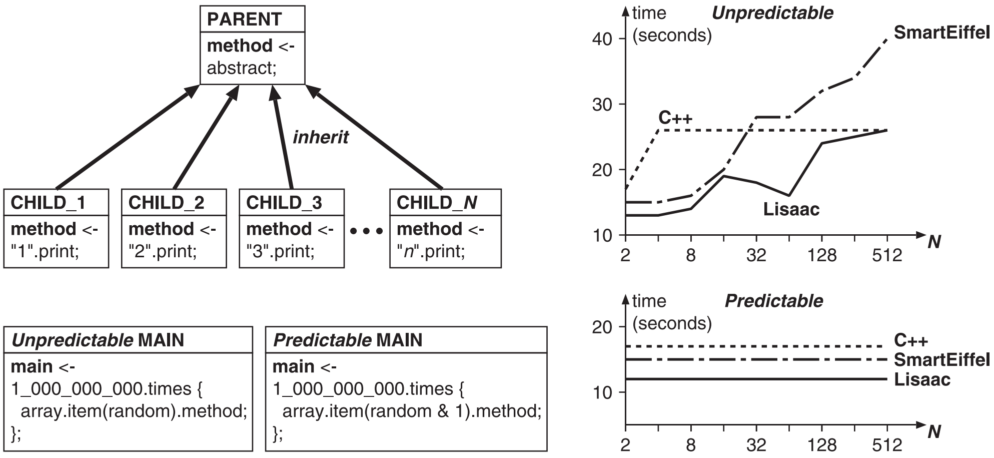


Figure 17. Lisaac^{SE} versus Lisaac^{LI}. The Lisaac^{SE} compiler is the early version of the Lisaac compiler written in SmartEiffel. The Lisaac^{LI} compiler is the first bootstrapped Lisaac compiler, written in Lisaac and recompiled with itself. As we applied a systematical translation of the SmartEiffel source code of Lisaac^{SE} to obtain Lisaac^{LI}, the comparison applies on the same algorithms.



```
method <-
abstract;
```

```
method <-
"1".print;
```

```
method <-
"2".print;
```

```
method <-
"3".print;
```

```
method <-
"n".print;
```

```
main <-
1_000_000_000.times {
  array.item(random).method;
};
```

```
main <-
1_000_000_000.times {
  array.item(random & 1).method;
};
```

Figure 18. Horizontal inheritance benchmark where N is the number of objects, from 2 to 512.

leading to the same code for each branch of the dispatching code. Such polymorphic method calls are replaced with direct static calls and accordingly considered as monomorphic call sites. Most of the extra monomorphic call sites are read/write operations of a field within a structure with the same displacement properties (details in [3]). The memory footprint gain, which is of 32.8% in Figure 17, is mostly due to the data structure compaction algorithm we had presented in Section 4.1. The run-time gain of 12.6% derives from better type analysis. Owing to this criteria, the Lisaac^{LI} compiler is more effective than the Lisaac^{SE} compiler.

DOI: 10.1002/spe

5.3. Late binding benchmarks, horizontal and vertical inheritance

Benchmarking only late binding appears to be quite difficult because of the involvement of several factors, such as the level of polymorphism, dynamic predictability of the target, or even the depth of the inheritance graph. Furthermore, most processor architectures are now using some sort of branch history table mechanism, which allows numerous conditional branches to be predicted when the same type of target is used repeatedly. Indeed, it is a well-known property of polymorphism that generally the receiver type at a polymorphic call site does not vary substantially [38]. Branch history table is thus used as a memory of the last receiver type, which may be considered as a type of inline caching performed by the processor (inline caching is an optimization technique that was first developed for Smalltalk [39, 40]).

The horizontal benchmark in Figure 18 is dedicated to late binding. Actually, this benchmark is not a single program but a set of programs depending on N, the number of subclasses stemming from the PARENT class. For instance, when N is 128, the PARENT class has 128 different subclasses with 128 different methods, and all of the 128 objects are mixed together into an array of 128 slots. Because N varies from 2 to 512, all types of polymorphism are considered. Furthermore, there are two different main programs, both having one polymorphic dispatch site embedded in a long-time loop. The *Unpredictable* `main` randomly shuffles the receiver amongst the N possibilities, making the dispatch method call truly unpredictable. Conversely, the *Predictable* `main` does not shuffle the receiver as much. Only two types of receivers are used repeatedly, making the predictable benchmark more reliable [41]. To have a meaningful comparison, the C++ code always uses the `virtual` keyword for all methods. There are only *virtual* methods both in SmartEiffel and Lisaac. Runtime results make it clear that the dispatch branching code is scalable in comparison with the VFT results of C++. In comparison with SmartEiffel, Lisaac is superior because the dispatch branching code itself is inlined. For an unpredictable situation, only C++ is time constant even when N becomes voluminous. With over 512 types to dispatch, even Lisaac is outperformed by C++; however, such a megamorphic call is clearly unrealistic (for megamorphic call sites, see [41–43]). Note that we have not mentioned the results of Java here because each Java virtual machine we have tried was outperformed. The performance time was between 30 and 50 times slower than C++ (in [44], they report a 10–100 factor compared with C).

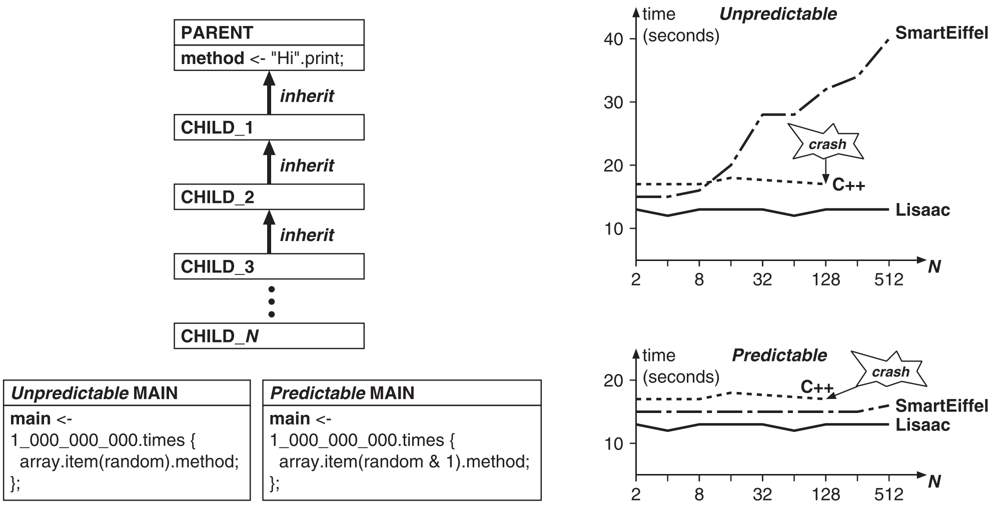
The vertical inheritance benchmark of Figure 19 also uses an N variable number of classes varying from 2 to 512 and, as in the previous benchmark, two different main programs, one unpredictable and one predictable. In the *vertical* benchmark, one unique method defined in the PARENT class is vertically inherited by all of the N subclasses. The C++ source code uses the `virtual` keyword. That is not that important, but the `gcc` compiler we used, `gcc 4.4.1`, was unable to compile more than 128 classes because of an internal limitation of the compiler for inheritance depth. The inadequate results of SmartEiffel in the unpredictable situation of Figure 19 is because the unique inherited method from PARENT is actually duplicated in each CHILD offshoot. SmartEiffel generates the dispatch branching code for the call. Lisaac is able to detect that the inherited method is always identical. All the dispatch branching code is thus avoided and the method is inlined. As in the horizontal benchmark, the C++ compiler generates a VFT for the call. The gains come only from the processor caching mechanisms.

5.4. Automatic cascading method calls

In Figure 20, the array accessed inside the loop contains 512 different instances of 512 different classes. The dispatch branching code is taken into account for each method call being performed on the same `receiver`; hence, it is unique (Section 3.6). Because of the VFTs, the C++ compiler is unable to efficiently take into account the dispatching code.

5.5. Call on the `self` (or the `this`) variable

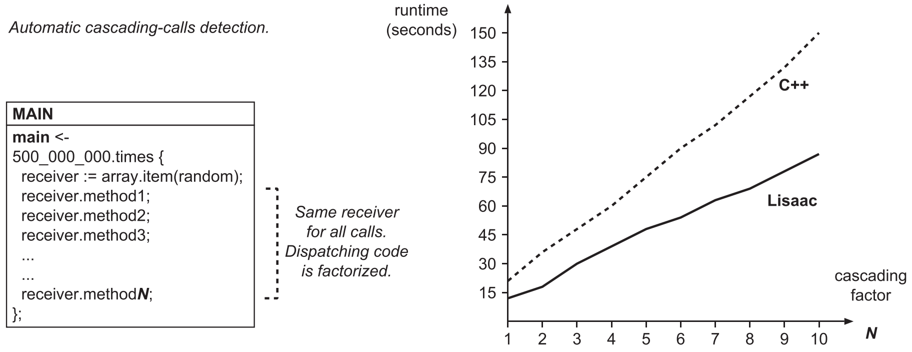
The purpose of the benchmark of Figure 21 is to measure the benefits obtained for method calls on the `self` (or the `this`) variable. In this family of programs, the number of method calls on `self` increases with N. Following the number of method calls on `self`, the number of defined subclasses



```
Unpredictable MAIN
main <-
1_000_000_000.times {
  array.item(random).method;
};
```

```
Predictable MAIN
main <-
1_000_000_000.times {
  array.item(random & 1).method;
};
```

Figure 19. Vertical inheritance benchmark where N is the number of objects, from 2 to 512.



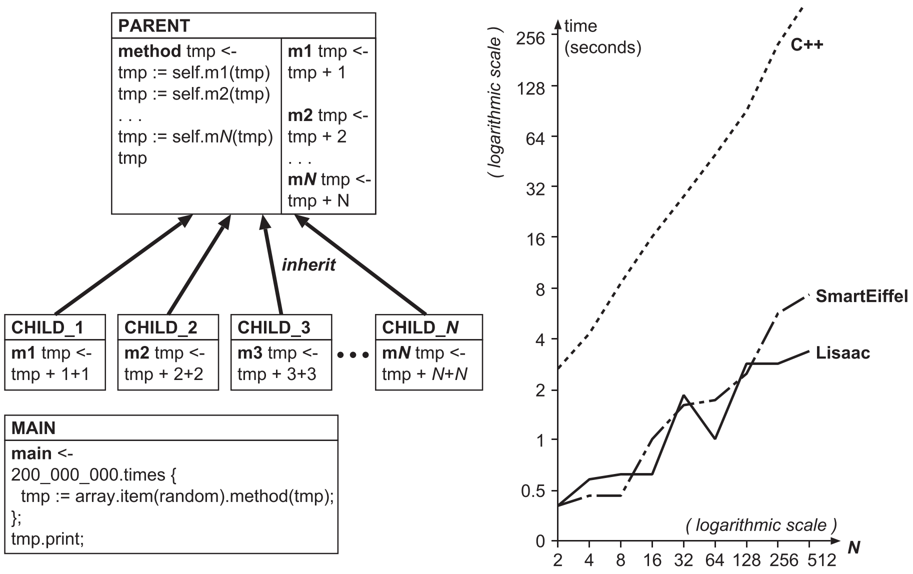
```
MAIN
main <-
500_000_000.times {
  receiver := array.item(random);
  receiver.method1;
  receiver.method2;
  receiver.method3;
  ...
  ...
  receiver.methodN;
};
```

Figure 20. Branch merging of Lisaac compared with C++. The slope variation highlights the gain.

increase to redefine inherited methods: CHILD_1 redefines only m1, CHILD_2 redefines only m2, CHILD_3 redefines only m3, and so on. The unpredictable main randomly shuffles the receiver among the N possibilities. As a result of the receiver customization technique previously presented in Section 3.2.1, both SmartEiffel and Lisaac have excellent results compared with C++. Actually, the execution time of C++ is so slow that we had to use a logarithmic scale. When N equals 256, Lisaac's runtime is 80 times shorter than the one for C++. Evidently, C++ treats method calls on the this receiver the same as for ordinary method calls, starting the dispatch process, again and again, from scratch.

5.6. Multiple inheritance

The purpose of the benchmark of Figure 22 is to measure the dynamic dispatch when multiple inheritance is involved. In this family of programs, the number of multiple inheritance links is increasing with N: GRANDCHILD_1 has 1 parent, GRANDCHILD_2 has 2 parents, GRANDCHILD_3 has 3 parents, and ... GRANDCHILD_N has N parents. The main loop randomly picks up a GRANDCHILD$_i$ from an array to launch the inherited method from PARENT. The C++ source code uses the virtual keyword both for inheritance and methods. For runtime results, even if SmartEiffel performs better (see upper-left curve of Figure 22), results of C++ are not completely unsatisfactory. We detected something never encountered in any previous benchmarks: the executable size of C++ grows dramatically with the value of N. To investigate this point, we measured the data, and the text segment of the generated executables for all values of N (see lower curves of Figure 22). As shown



```
PARENT
method tmp <-
tmp := self.m1(tmp)
tmp := self.m2(tmp)
. . .
tmp := self.mN(tmp)
tmp

m1 tmp <-
tmp + 1
m2 tmp <-
tmp + 2
. . .
mN tmp <-
tmp + N

CHILD_1
m1 tmp <-
tmp + 1+1

CHILD_2
m2 tmp <-
tmp + 2+2

CHILD_3
m3 tmp <-
tmp + 3+3

CHILD_N
mN tmp <-
tmp + N+N

MAIN
main <-
200_000_000.times {
  tmp := array.item(random).method(tmp);
};
tmp.print;
```

Figure 21. Call on `self` / `this` inside a template method pattern [31].

in the figure, it appears that the problem of C++ arises in the text segment part of the executable, clearly indicating that the tables used for the multiple parents are growing larger with the value of N (this confirms the results mentioned in [45]).

5.7. *Impact of the inlining level of small functions*

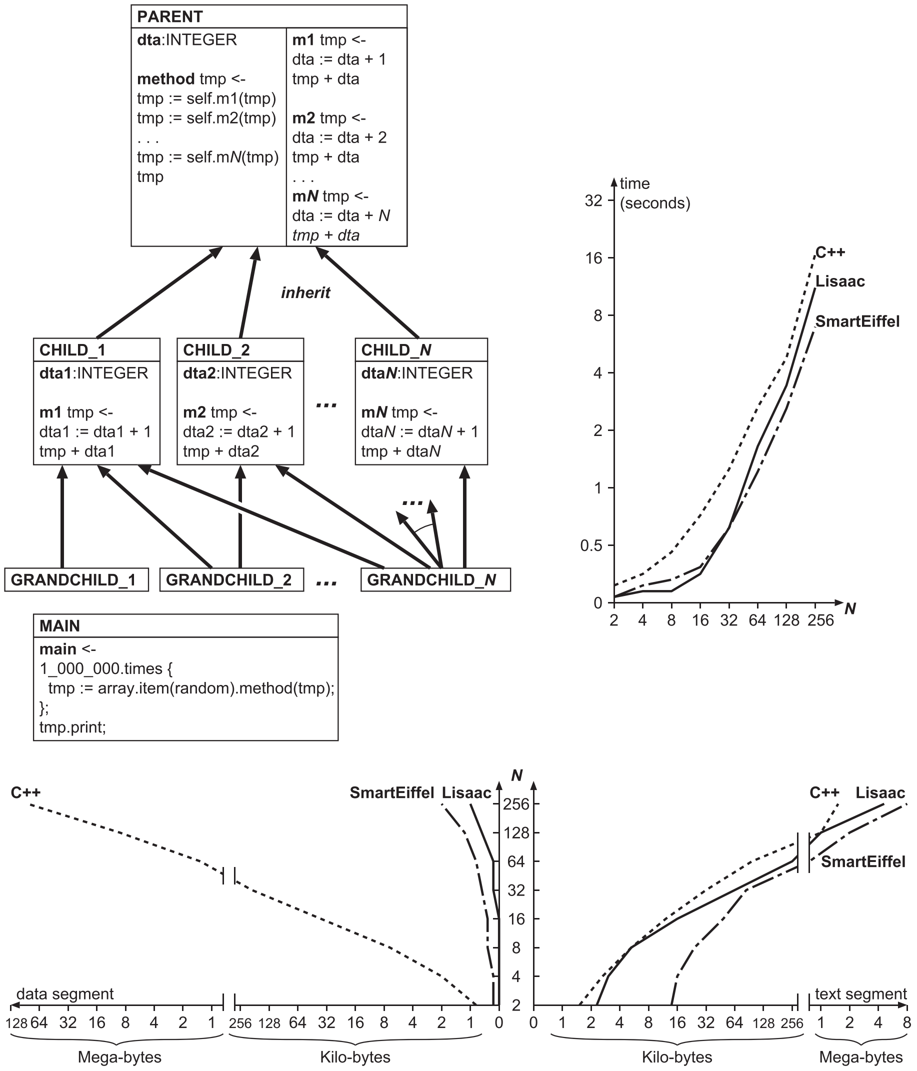
The inlining level of small functions (i.e. case #5 of our inlining strategy described in Section 3.3) can be selected manually, thanks to a compiler option. Figure 23 shows the impact of the inlining level on binary code size while compiling the Lisaac compiler itself. After reaching a minimum inlining level of 10, the curve increases linearly. Because the Lisaac compiler is written in an object-oriented way and contains numerous methods with several polymorphic method calls, it is quite likely that several other object-oriented applications will have a similar behavior. Thus, we decided that 10 is a good trade-off for the inlining level default value.

5.8. *The MPEG2 benchmark*

To have another significant benchmark, we translated an entire $Mpeg2$ decoder, originally written in C, into Lisaac. We performed a mechanical translation of the original C code, approximately 10,000 lines of C code. To have a significant runtime, we used an 80 Mb video file as input. Four versions were created, one for each of the following output formats: YUV, SIF, TGA, and PPM. Figure 24 shows the runtime of the C code compared with the Lisaac code. The C code generated by the Lisaac compiler was compiled with the same C compiler and options. As for the YUV conversion, Lisaac is much slower than C. Amazingly, all the other conversions (SIF, TGA, and PPM) require to first compute the YUV conversion. The time used for the YUV conversion is indicated on each conversion to exhibit the remarkable results for the remainder of the computation (Figure 24).

6. RELATED WORK

MLton (http://www.mlton.org/) is an open-source, whole-program, optimizing Standard ML compiler. Regardless of the fact that MLton is for the SML functional language (i.e. nonobject oriented), MLton is very similar to Lisaac: CWA, aggressive dead code elimination, untagged and unboxed native integers, unboxed native arrays, and so on. Furthermore, MLton is very similar to Lisaac in the way closures are inlined.



```
PARENT
dta:INTEGER

method tmp <-
tmp := self.m1(tmp)
tmp := self.m2(tmp)
. . .
tmp := self.mN(tmp)
tmp

m1 tmp <-
dta := dta + 1
tmp + dta

m2 tmp <-
dta := dta + 2
tmp + dta
. . .
mN tmp <-
dta := dta + N
tmp + dta
```

```
CHILD_1
dta1:INTEGER

m1 tmp <-
dta1 := dta1 + 1
tmp + dta1
```

```
CHILD_2
dta2:INTEGER

m2 tmp <-
dta2 := dta2 + 1
tmp + dta2
```

```
CHILD_N
dtaN:INTEGER

mN tmp <-
dtaN := dtaN + 1
tmp + dtaN
```

```
MAIN
main <-
1_000_000.times {
  tmp := array.item(random).method(tmp);
};
tmp.print;
```

Figure 22. Multiple inheritance benchmark where N is the number of multiple inheritance links.

Work presented in [46] introduce the concepts of type flow analysis and detail its use in reducing runtime overhead in Oberon-2 [47]. Their main goal is to eliminate irrelevant dynamic type tests. The type flow analysis of Lisaac is notably more complete to handle all kinds of variables of object-oriented languages (see Section 2.2). Furthermore, we introduce in Section 2.3 an innovative approach to handle type of elements within arrays. This is a key point of our strategy because it allows us to perform a real global program analysis.

The Vortex compiler [48] is a language-independent optimizing compiler for object-oriented languages. There are front-ends for Cecil, C++, Java, and Modula-3. Vortex is an excellent tool to quantify the benefits of object-oriented optimizations. Actually, most Vortex optimizations are present in the Lisaac compiler, and the type flow analysis of Lisaac is both intraprocedural and interprocedural. Vortex provides selective recompilation, whereas Lisaac does not. Rather than specializing exhaustively, Vortex is guided by dynamic profile data to selectively specialize only heavily used methods. Techniques in Vortex such as profile-guided optimizations and selective recompilation might be profitably added to our compilation strategy.

The type inference carried out for the Self language in [49] is similar to the Lisaac approach: each expression is considered separately. Their algorithm does not work on the full transitive closure graph but on a fragment only. The code generation is then performed by using this fragment

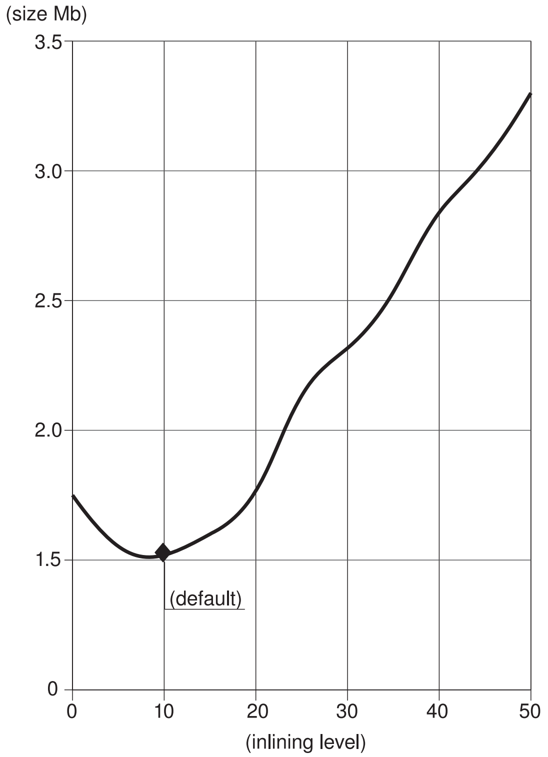


Figure 23. Impact of the inlining level on the binary size of the Lisaac compiler itself.

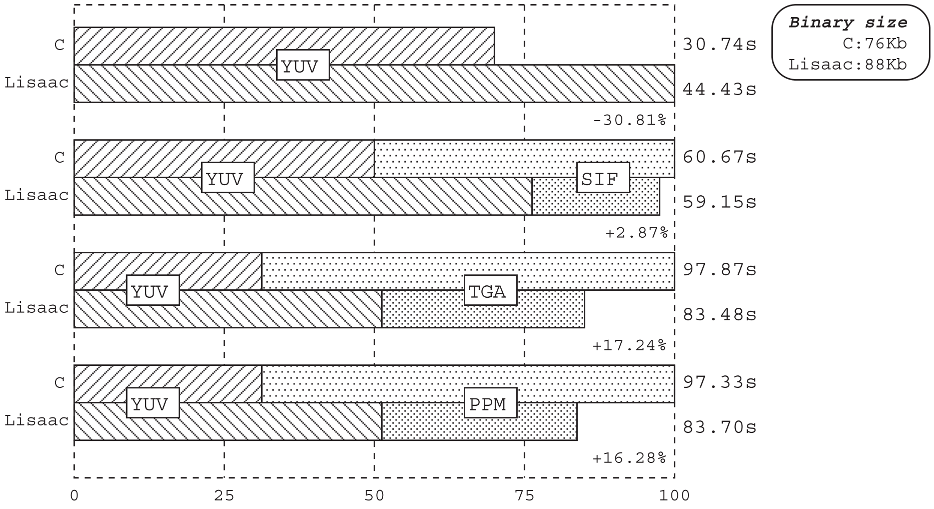


Figure 24. MPEG2 runtime benchmark.

before doing an inference on a larger fragment. This method is less expensive in terms of memory because it is an incremental process. On the other hand, the number of possible dynamic type for one call site is never static, even at code generation time. Our approach requires a superset of all the possible dynamic types for each method call.

Marmot [12] is an optimizing compiler for a large subset of Java. Marmot is intended primarily as a high-quality research platform. Marmot performs a class hierarchy analysis [28] and a complete program analysis. It takes verified bytecode as input instead of Java source code, and it still uses VFT. Marmots object-oriented optimizations are implemented using a combination of intermodule

DOI: 10.1002/spe

flow-insensitive and per-method flow-sensitive techniques. Contrary to our methods, flow analysis is not globally performed. The authors also indicate that optimum performance is best achieved under the CWA. The stack allocation optimization of Marmot improves locality and reduces garbage collection overhead by allocating objects with bounded lifetimes on the stack rather than on the heap. Such a stack allocation should be added in our compilation strategy. Marmot offers a choice of three garbage collection schemes: a conservative collector, a copying collector, and a generational copying collector.

As presented in [50], staying type-safe when inlining a virtual method may cause problems between the receiver's static and dynamic type. However, with our approach, the dynamic type dispatching is realized before inlining, and thus, the problem does not even exist. In [51], a global program analysis with type inference is performed on a Smalltalk like language. Yet, the graph contains all the type information that can be derived from the program without keeping track of NULL values or flow analyzing the contents of the instance variables. The collected information is close to the RTA algorithm applied on static type languages.

The Fiji VM [44] compiles Java bytecode to C for Embedded Hard Real-Time Devices also using the CWA. Fiji features many similarities with Lisaac: type propagation both intraprocedural and interprocedural using a static single assignment intermediate representation. In addition, Fiji takes into account multithreading with a concurrent real-time garbage collector. The major difference is that Fiji still uses VFTs.

7. FUTURE WORK

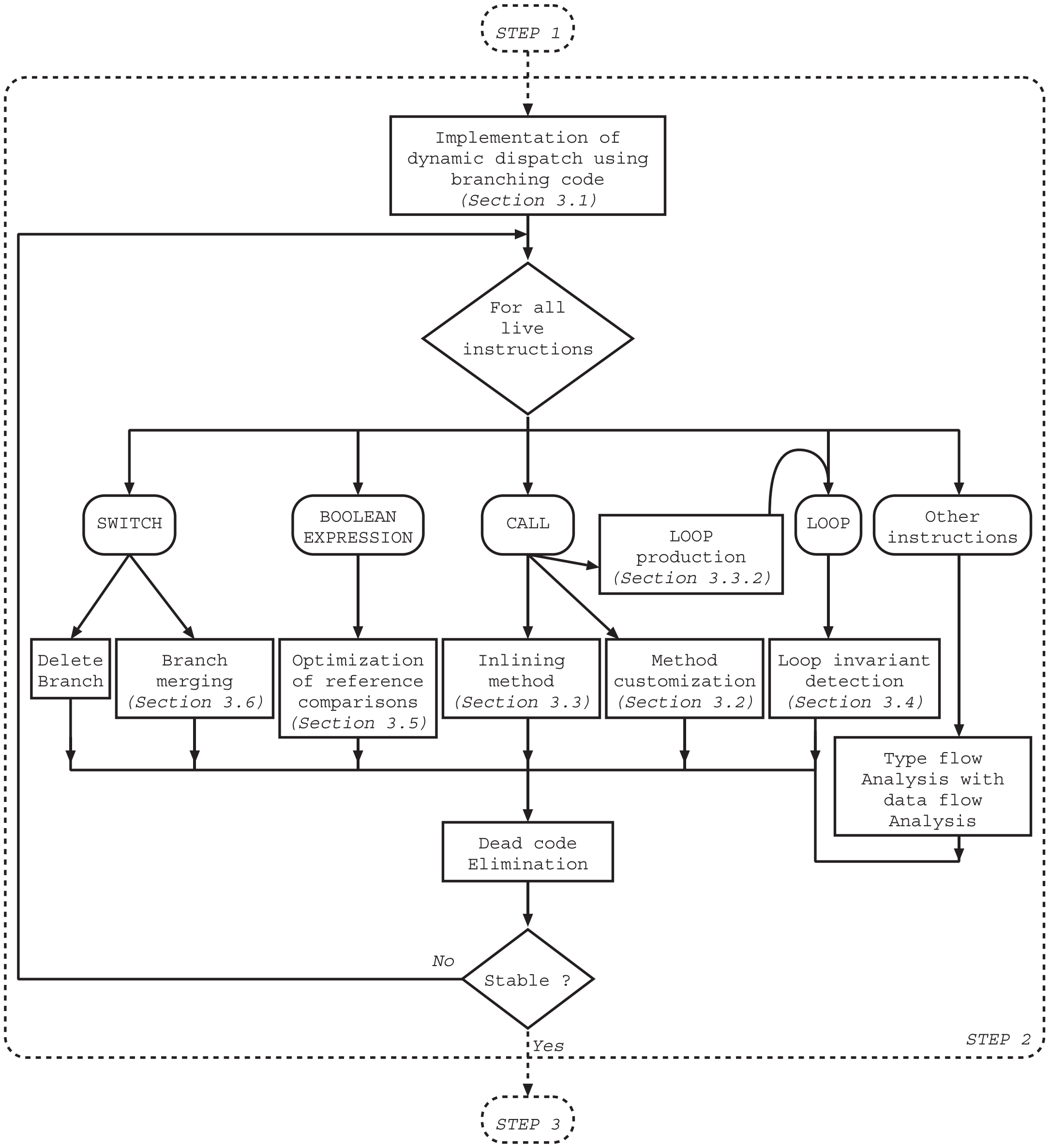
We have learned from Eiffel the fundamental aspects of design by contract to make the code more readable, to validate it, and to ease debugging. We are now experimenting with design by contract as a possible source of information for the compiler to optimize the code. Roughly, the main idea is to make the assumption that preconditions, postconditions, and invariants that are written by the programmer are always correct and may be trusted by the compiler. We hope that combining the information gathered by data/type flow analysis will allow better code optimization. When considered valid, design by contract information could be used to perform high-level optimizations. Static detection of contract violations is also something we want to investigate. While the distribution of computation over the network can be reasonably achieved, thanks to library support, efficient access to local multicore power is an important point we need to address. We are currently working on a brand new concurrency model, compatible with our compilation strategy, also using CWA, allowing safe usage of multithreading.

8. CONCLUSION

The compilation strategy we presented is the result of a long project on two real-size compilers, SmartEiffel being historically the precursor of the advanced compilation strategy of Lisaac. Whereas both compilers work under CWA, Lisaac adds type flow analysis, also looking *inside* arrays with a simple technique (Section 2.3). As a consequence, the type flow analysis is not blocked while reading references of objects from arrays, making a truly global type flow analysis possible. Global type flow analysis combined with code customization allows us to predict the dynamic type of numerous method calls. For instance, in the whole Lisaac compiler, 98% of method calls are statically resolved and replaced with static calls (Section 5.2). To tackle the explosion of code size, we set up our ATS method customization strategy together with transformation rules inside the dispatch branching code (Sections 3.2.3 and 3.3). As an important result, inlining of closures as well as tail recursion removal allows a perfect translation of the library-defined control statements (Sections 3.3.1 and 3.3.2). Our compilation strategy could be used for most object-oriented, class-based or prototype-based languages, assuming the availability of the whole source code of the application. Throughout the article, compiler writers may find useful measurements to guide their decisions when they have to choose amongst optimizations.

DOI: 10.1002/spe

APPENDIX A: FOCUS ON STEP 2


STEP 1
Implementation of dynamic dispatch using branching code (Section 3.1)
For all live instructions
SWITCH
BOOLEAN EXPRESSION
CALL
LOOP production (Section 3.3.2)
LOOP
Other instructions
Delete Branch
Branch merging (Section 3.6)
Optimization of reference comparisons (Section 3.5)
Inlining method (Section 3.3)
Method customization (Section 3.2)
Loop invariant detection (Section 3.4)
Type flow Analysis with data flow Analysis
Dead code Elimination
No
Stable ?
Yes
STEP 2
STEP 3

ACKNOWLEDGEMENTS

The authors wish to thank Xavier Oswald for contributing to this project with insightful discussions, ideas, and programming support of some benchmarks. We also thank Jean-Pierre Camal and Vasilica Le Floch for their many helpful suggestions. Pierre-Alexandre Voye also helped us clarify some parts of the article. We thank Matthieu Herrmann, Philippe Ribet, Cyril Adrian, Nicolas Boulay, Claire Quirke, and Karen Fournier. We also thank the anonymous referees for their helpful comments.

REFERENCES

1. Sonntag B, Colnet D. Lisaac: the power of simplicity at work for operating system. In *40th Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'2002)*. Australian Computer Society: Sydney, Australia, 2002; 45–52.
2. Sonntag B, Colnet D, Zendra O. Dynamic inheritance: a powerful mechanism for operating system design. *'Intercontinental Workshop on Object-Orientation and Operating Systems (OOOSWS'2002)' – ECOOP'02 Workshop Reader – Málaga*, Spain, 2002; 25–30.

3. Zendra O, Colnet D, Collin S. Efficient dynamic dispatch without virtual function tables. The smalleiffel compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. ACM Press: Atlanta, Georgia, 1997; 125–141.
4. Goldberg A, Robson D. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading: Massachusetts, 1983.
5. Ungar D, Smith RB. Self: the power of simplicity. In *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*. ACM Press: Orlando, Florida, 1987; 227–241.
6. Chambers C. *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Department of Computer Science of Stanford University: California, 1992.
7. Chambers C, Ungar D. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *4th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*. ACM Press: New Orleans, Louisiana, 1989; 146–160.
8. Dixon R, McKee T, Schweitzer P, Vaughan M. A fast method dispatcher for compiled languages with multiple inheritance. In *4th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*. ACM Press: New Orleans, Louisiana, 1989; 211–214.
9. Vitek J, Horspool R. Taming message passing: efficient method look-up for dynamically typed languages. In *8th European Conference on Object-Oriented Programming (ECOOP'94)*. Springer-Verlag: Bologna, Italy, 1994; 432–449. LNCS 821.
10. Grove D, Chambers C. A framework for call graph construction algorithms. *ACM Transaction on Programming Languages and Sytems* 2001; **23**(6):685–746.
11. Zibin Y, Gil J. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *17th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'2002)*. ACM Press: Seattle, Washington, USA, 2002; 142–160.
12. Fitzgerald R, Knoblock TB, Ruf E, Steesgaard B, Tarditi D. Marmot: an optimizing compiler for Java. *Software Practice & Experience* 2000; **30**(3):199–232.
13. Collin S, Colnet D, Zendra O. Type inference for late binding. the SmallEiffel compiler. In *Proceedings of the Joint Modular Languages Conference*, Vol. 1204, Lecture Notes in Computer Sciences, Lintz, Austria, 1997; 67–81.
14. Colnet D, Coucaud P, Zendra O. Compiler support to customize the mark and sweep algorithm. *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)* 1999; **34**(4):154–165.
15. Zendra O, Colnet D. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software Practice & Experience* 2001; **31**(6):601–613.
16. Futamura Y. Partial evaluation of computation process – an approach to a compiler–compiler. *Reprinted in Higher-Order and Symbolic Computation* 1999; **12**(4):381–391.
17. Consel C, Danvy O. Tutorial notes on partial evaluation. *20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, 1993; 493–501.
18. Ducournau R, Morandat F, Privat J. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In *24th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'09)*. ACM Press: Orlando, Florida, USA, 2009; 41–60.
19. Bacon DF, Sweeney PF. Fast static analysis of C++ virtual function calls. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, Vol. 31(10). ACM Press: San Jose, California, 1996; 324–341.
20. Neil D. Jones flow analysis of lambda expressions. *Lecture Notes in Computer Science, Automata, Languages and Programming* 1981; **115**:114–128.
21. Shivers O. Control-flow analysis in scheme. In *Conference on Programming Language Design and Implementation (PLDI)*, Vol. 23(7). ACM SIGPLAN Notices: Atlanta, Georgia, 1988; 164–174.
22. Blanchet B, Cousot P, Feret J, Mauborgne L, Min A, Monniaux D, Rival X. A static analyser for large safety-critical software. In *Conference on Programming Language Design and Implementation (PLDI)*, Vol. 38. ACM SIGPLAN Notices: San Diego, California, 2003; 196–207.
23. Jones R, Lins R. *Garbage Collection*. Wiley, 1996. ISBN 0-471-94148-4w.
24. Colnet D, Coucaud P, Zendra O. Compiler Support to Customize the Mark and Sweep Algorithm. *In International Symposium on Memory Management (ISMM'98)*, Vancouver, Canada, 1998; 154–165.
25. Colnet D, Sonntag B. *Paper in French: Analyse simple de types dans les tableaux et optimisation du ramasse-miettes*. *Conférence en IngénierieE du Logiciel (CIEL'12)*, Rennes, France. Available from: http://gpl2012.irisa.fr/sites/default/files/CIEL2012-Colnet-paper17.pdf.
Conférence en IngénierieE du Logiciel (CIEL'12), Rennes, France
26. Ellis MA, Stroustrup B. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading: Massachusetts, 1990.
27. Driesen K, Hölzle U. The direct cost of virtual function calls in C++. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, Vol. 31(10). ACM Press: San Jose, California, 1996; 306–323.
28. Dean J, Grove D, Chambers C. Optimization of object-oriented programs using static class hierarchy analysis. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*. Springer-Verlag: Åarhus, Denmark, 1995; 77–101. LNCS 952.

DOI: 10.1002/spe

29. Agesen O. The cartesian product algorithm: simple and precise type inference of parametric polymorphism. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*. Springer-Verlag: Åarhus, Denmark, 1995; 2–26. LNCS 952.
30. Shao Z, Appel AW. Space-efficient closure representations. In *ACM Conference on Lisp and Functional Programming (ICFP'1994)*. ACM Press: Orlando, Florida, 1994; 150–161.
31. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading: Massachusetts, 1995.
32. André P, Royer J-C. Optimizing method search with lookup caches and incremental coloring. In *7th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*, Vol. 27(10). ACM Press: Vancouver, British Columbia, Canada, 1992; 110–127.
33. Driesen K. Selector table indexing and sparse arrays. In *8th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, Vol. 28(10). ACM Press: Washington, D.C., 1993; 259–270.
34. Driesen K, Hölzle U. Minimizing row displacement dispatch tables. In *10th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, Vol. 30(10). ACM Press: Austin, Texas, 1995; 141–155.
35. Palacz K, Vitek J. Java subtype tests in real-time. In *17th European Conference on Object-Oriented Programming (ECOOP'03)*. Springer-Verlag: Darmstadt, Germany, 2003; 378–404.
36. Ducournau R. Perfect hashing as an almost perfect subtype test. *ACM Transaction on Programming Languages and Sytems* 2008; **30**(6):1–56.
37. Lee H, Dincklage D, Diwan A, Eliot Moss JB. Understanding the behavior of compiler optimizations. *Software Practice & Experience* 2006; **36**(8):835–844.
38. Hölzle U. Adaptative optimization for Self: reconciling high performance with exploratory programming. *Ph.D. Thesis*, Stanford University, 1994. CS-TR-94-1520.
39. Deutsch PL, Schiffman A. Efficient implementation of the smalltalk-80 system. *11th Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, UT; 1984.
40. Ungar D, Patterson D. What Price Smalltalk? *IEEE Computer* 1987; **20**(1):67–74.
41. Driesen K, Hölzle U, Vitek J. Message dispatch on pipelined processors. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*. Springer-Verlag: Åarhus, Denmark, 1995; 253–282. LNCS 952.
42. Hlzle U, Chambers C, Ungar D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *5th European Conference on Object-Oriented Programming (ECOOP91)*, Vol. 512. Springer-Verlag: Geneva, Switzerland, 1991; 21–38.
43. Aigner G, Hölzle U. Elimination virtual function calls in C++ programs. In *10th European Conference on Object-Oriented Programming (ECOOP'96)*. Springer-Verlag: Linz, Austria, 1996; 142–166. LNCS 1098.
44. Pizlo F, Ziarek L, Blanton E, Maj P, Vitek J. High-level programming of embedded hard real-time devices. In *5th European Conference on Computer systems (EuroSys'10)*. ACM SIGOPS: Paris, France, 2010; 69–82.
45. Privat J, Ducournau R. Link-time static analysis for efficient separate compilation of object-oriented languages. In *6th Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*. ACM SIGPLAN-SIGSOFT: Lisbon, Portugal, 2005; 20–27.
46. Corney D, John Gough J. Type test elimination using typeflow analysis. In *Proceedings of Programming Languages and System Architectures*, Vol. 792, Lecture Notes in Computer Sciences. Springer-Verlag, 1994; 137–150.
47. Mössenböck H, Wirth N. The Programming Language Oberon-2. *Computer Science Report 160*, ETH Zurich, May 1991.
48. Dean J, DeFouw G, Grove D, Litvinov V, Chambers G. Vortex: an optimizing compiler for object-oriented languages. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, Vol. 31(10). ACM Press: San Jose, California, 1996; 83–100.
49. Agesen O, Palsberg J, Schwartzbach MI. Type inference of SELF: analysis of objects with dynamic and multiple inheritance. *Software Practice & Experience* 1995; **25**(9):975–995.
50. Glew N, Palsberg J. Type-safe method inlining. In *16th European Conference on Object-Oriented Programming (ECOOP'02)*. Springer-Verlag: Málaga, Spain, 2002; 525–544.
51. Palsberg J, Schwartzbach MI. Object-oriented type inference. In *6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, Vol. 26(11). ACM Press: Phoenix, Arizona, 1991; 146–161.

DOI: 10.1002/spe