SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **0**:1–12 *Prepared using* speauth.cls *[Version: 2000/03/06 v2.1]*

Coping with aliasing in the GNU Eiffel Compiler implementation

Olivier ZENDRA* and Dominique COLNET

LORIA, UMR 7503 (INRIA - CNRS - Univ. of Nancy), BP 101, 54602 Villers-lès-Nancy Cedex, FRANCE
E-mail: {zendra,colnet}@loria.fr
http://SmallEiffel.loria.fr

SUMMARY

This paper reports our experience about aliasing in the implementation of SmallEiffel, the GNU Eiffel Compiler. The SmallEiffel compiler source code makes intensive use of aliasing in order to achieve very good performance. We explain how aliasing can be handled in a safer way, and how helpful the design by contract capabilities of the Eiffel language have been. The Singleton pattern appears to be crucial in implementing alias provider objects. We propose an efficient implementation of this pattern made easy by some Eiffel idioms. This technique, which requires no language modification, is very appropriate for compilation, but can also be applied to a much wider range of applications. We show the performance impact of string aliasing both in terms of memory footprint and execution time.

KEY WORDS: aliasing, performance impact, design by contract, Eiffel, SmallEiffel, assertions, Flyweight pattern, Singleton pattern

1. INTRODUCTION

As soon as there is more than one reference to the same object at the same time, this object is said to be *aliased*. This phenomenon is named *aliasing* and has been known for a long time (see for example [1, 2]).

It is sometimes considered as a dangerous feature which should be avoided as much as possible in programs. Indeed, aliasing does make programs more difficult to debug, since in a certain way it can be said it breaks encapsulation: an object `o1` may modify the state of an object `o2` it points to without notifying other objects `o3`, `o4`, ... referencing the same aliased

*Correspondence to: Olivier Zendra, LORIA - INRIA Lorraine, BP 101, 54602 Villers-lès-Nancy Cedex, FRANCE

object o2, thus causing unexpected state changes from o3, o4, ... point of view. Some work has thus been focused on alias detection, prevention and avoidance, for software engineering purposes.

It is however quite difficult, if not impossible, to avoid aliasing, especially in object-oriented languages. Furthermore, aliasing, especially because it means sharing objects, can be convenient and offers potential optimizations in terms of memory as well as speed. Techniques have thus been sought to control aliasing and prevent its potential nuisances, while reaping its benefits.

We strongly believe aliasing offers some significant advantages in terms of performance and thus is definitely worth building on. For these reasons, the implementation of our Eiffel [3] compiler — SmallEiffel, The GNU Eiffel Compiler† — makes intensive but carefully designed use of aliasing, trying to combine performance and safety through use of techniques which are mostly already known.

The use of aliasing is of course not limited to compilers, but can be applied to most kinds of software. Graphical user interfaces (GUI) are a domain where aliasing is intensive. As an example, the famous MVC (Model-View-Controller) model, initially developed for Smalltalk-80 [4] and now used in a wide range of systems, can even be said to be based on aliasing. Indeed, it requires one unique model of the world to be referenced (hence aliased) by all views of this model. Using several instances — or copies — of the model would be a nonsense, because it would bring up severe coherency issues between them.

Obviously, aliasing is also present in many other domains, including search engines, pattern matching, interpreters, compilers, databases, etc. Our own work focussed on aliasing in a compiler, and this paper thus reports on our experience and results with aliasing within the SmallEiffel compiler implementation. We explain how to better control the propagation of aliases, thanks to design by contract [2]. We show how Eiffel assertions were especially suited for this, although other languages also offer mechanisms which can be used to better control alias propagation. Since the Singleton pattern [5] was needed to control alias production and propagation, we provide an Eiffel idiom efficiently implementing this pattern.

The remainder of this paper is organized as follows. In section 2, we first remind why aliasing can be considered especially useful in a compiler. Section 3 then introduces the concept of *alias provider* and details how it can be implemented in Eiffel by using the Singleton pattern. Then we describe in section 4 how design by contract and Eiffel assertions can be used to enforce a safer use of aliasing. Section 5 considers the problem of mutable object aliasing. Section 6 presents performance measurements and discusses the benefits of using aliasing in SmallEiffel. Finally, section 7 reviews some related work and section 8 concludes.

†Note that SmallEiffel, the GNU Eiffel Compiler, is itself implemented in Eiffel (*bootstrapped*) and is available with its source code and libraries at `http://SmallEiffel.loria.fr`.

2. ALIASING: WHAT FOR, IN A COMPILER?

As we mentioned, aliasing is of course not specific to compiler implementations. However, compilers appear to be prone to using aliasing, for example when encountering an identifier.

One of the common tasks of a compiler is for example to check whether a variable is a local one, an argument of the enclosing routine, a global variable, or even some undeclared identifier. To efficiently perform this task, one must be careful to avoid costly string content comparisons. Indeed, it is extremely frequent in a compiler to need to know whether two symbols are the same or not.

The basic idea of aliasing here is to use exactly the same string when several symbols have the same name, *à la* Flyweight pattern [5]. This way, it is possible to compare two symbol names with a mere pointer comparison (operator `=` in Eiffel, equivalent to `==` in C), instead of having to potentially look at the complete contents of the name (with `is_equal`, the Eiffel equivalent of `strcmp` in C).

As a consequence, there seems to be a possible gain in terms of speed. Duplicates of the same string are also avoided, which saves a lot of memory. However, the cost for managing such a symbol name aliasing system may offset part of its advantages and thus has to be more precisely evaluated.

Of course, strings are not the only "aliasable" objects in a compiler. However, because strings are very common and well understood objects, we will use them as a basic example all along this paper. Many other kinds of more complex objects are good candidates for aliasing, such as for example the various objects used to represent abstract syntax tree [6], or AST, nodes.

3. USING AN ALIAS PROVIDER

Shared symbol names are akin to the *symbols* provided as early as 1960 by Lisp [7] or in the 80's by Smalltalk [8], or to the 90's *interned strings* of Java [9]. This kind of mechanism is not provided by the Eiffel language. Eiffel developers thus have to write their own code to explicitly take care of interning, or aliasing.

In SmallEiffel, we implemented this kind of aliasing with an *alias provider* (or *aliaser*). Indeed, although the management of locally aliased objects is quite easy and intuitive in a routine or small algorithm, this is not the case anymore when aliasing has to be managed on a system-wide basis.

In a compiler for example, although the symbol names are created in the parser only, they are handled by a special object in charge of providing a unique, shared object for each name. This special object is called in SmallEiffel and in this paper the STRING_ALIASER. It is not dissimilar to a *bridge* giving access to *islands* of objects [10]. However unlike the bridge object our alias provider is not intended to give only temporary access to the objects it holds (*dynamic* aliasing [11]) but also a controlled, long-term access (*static* aliasing).

To achieve its purpose, it is above all very important that the STRING_ ALIASER object be unique in the whole system. The Singleton pattern [5] perfectly suits this purpose.

3.1. Implementing the Singleton pattern in Eiffel

The role of the Singleton pattern is to ensure a given class only has one canonical instance and to provide global access to it.

The proposed C++ implementation in [5] guarantees that the Singleton class is instantiated only once by using an explicit test instruction at runtime.

In Java it is possible to ensure with no runtime cost that only one instance of a class is ever created. This is done by making the constructor private, creating a single static instance in the static initialization block and redefining the `clone` method so that it returns a reference to the static instance.

We propose a novel way of implementing the Singleton pattern in Eiffel, by relying on the assertion and *once* mechanisms, as follows:

```
class A_SINGLETON

 -- Other features here...

feature {NONE}  -- Singleton part

   singleton_memory: A_SINGLETON is
      once
         Result := Current;
      end;

invariant

   is_actually_singleton: Current = singleton_memory

end  -- A_SINGLETON
```

`Current` is the Eiffel notation for the current object, like `self` in Smalltalk [8] or `this` in Java [9] or C++ [12].

`singleton_memory` is an Eiffel `once` function [3]. Such a function is executed only once, the first time it is called. Subsequent calls return exactly the same result which is automatically memorized for all instances of the same class. Since `singleton_memory` is exported to NONE, it is private and can only be called from objects of type A_SINGLETON or its heirs. This way, the class function `singleton_memory` memorizes the (reference to the) first object that calls it.

Invariants in Eiffel are assertions that are triggered each time a client calls a feature on an instance of the class enclosing the invariant. Invariants are inherited by descendants of the class they appear in. The `is_actually_singleton` invariant ensures that only one object of type A_SINGLETON (or subtype thereof) is ever created in the system. Indeed, it checks that the (reference to the) `Current` object is exactly the one stored by `singleton_memory`. If more than one object of type A_SINGLETON were to be created, this invariant would fail and trigger an exception, because Current would refer to the second object, whereas singleton_memory would refer to the first.

Is is important to note that like the Java solution mentioned above, the Eiffel implementation of Singleton incurs no overhead at all in optimized mode for application delivery. Indeed, invariants, like other assertions, are tools designed to be used in the development phase of the code to ease debugging and document the code. In the delivery code, they are not activated anymore.

Of course, because of the semantics of Eiffel once routines, each class which must be a singleton has to implement the singleton_memory function and is_actually_singleton invariant. Inheriting from A_SINGLETON is not a viable solution, because this way it would be impossible to have more than one singleton in the system.

3.2. Implementation of an alias provider

Given the above implementation for singletons, the specification of our alias provider is quite simple. In addition to the code for the Singleton pattern, our STRING_ALIASER offers simply one function, item, in charge of providing a reference to the canonical representation for the specified model (like the intern function of Java class String). Here is its interface:

```
class interface  STRING_ALIASER

   item (model: STRING): STRING

      require
         model /= Void

      ensure
         Result.is_equal(model)

invariant
   is_actually_singleton: Current = singleton_memory
```

The first time item is called with a certain name as an argument, it returns it and memorizes the model string. Subsequent calls with either the same model string or different model strings with the same contents will all return the (reference to) the very first, memorized, canonical string object.

The precondition (keyword require) of function item checks that the argument is not Void[‡]. The postcondition (ensure) verifies that the returned object has the same content as the argument. As we have previously explained for the invariant, preconditions and postconditions are not a proof of correctness. They are a validation tool used during development. They thus incur no overhead at runtime in optimized mode. Note that in other languages, the simple assertions above can be replaced by mechanisms like assert in C, C++ or Java.

[‡] Void is the Eiffel equivalent of NULL in C or null in Java.

Internally, the memorization is done within `storage`, a private structure in the STRING_ALIASER. This `storage` is an instance of DICTIONARY, a kind of hash table, managed similarly to the one provided by the C++ STL (Standard Template Library, [13]). Note that unlike Smalltalk or Java, Eiffel does not provide a hash code for each object. Classes whose instances have to be hashed need to explicitly inherit from HASHABLE. This is the case of the standard class STRING as per the ELKS'95 (Eiffel Kernel Library Standard) definition [14]. The hash code for SmallEiffel STRINGs is defined as $\sum_{c=1}^{count} ascii(c) * 5^{c-1}$, where c is the character rank in the string, 1 the index of the first character and *count* the index of the last one.

4. ENFORCING THE ALIAS PROVIDER USE

We have described in the previous sections how the uniqueness of the alias provider could be guaranteed. It is of course also very important that in the whole system all objects use the alias provider, and that none keeps its own duplicate versions of aliased strings. This way only one single, unique version of the string is ever created and is managed by the alias provider.

So all classes of the system have to respect the rules of the game, and use the alias provider. They also have to take part in the whole system security by checking that their providers themselves do use the alias provider. This can be easily done in Eiffel by using the preconditions and postconditions of routines.

The following example features LOCAL_NAME, a class used by SmallEiffel to represent an identifier corresponding to the name of a local variable. The creation routine `make` takes two arguments: one to record the position in the source code, the other for the string representing the identifier name.

```
class interface LOCAL_NAME

   --
   -- A local name in some declaration list.
   --

creation

    make (sp: POSITION; n: STRING)
       require
          sp /= Void;
          n = string_aliaser.item(n)  -- (#1)
       ensure
          start_position = sp;
          to_string = n               -- (#2)
```

In the above code, assertion *(#1)* is a precondition which checks that the caller is passing an *aliased* string argument generated by the STRING_ALIASER. Indeed, if the contents of `n` is the same as the one of a string memorized by the string aliaser, `string_aliaser.item` returns the memorized STRING object. Thus, if STRING `n` has been provided by the STRING_ALIASER, the memorized object corresponding to the contents of `n` is `n` itself.

Similarly, postcondition *(#2)* also guarantees the correct behavior of the `make` routine, that is that the aliased string has been memorized by the current LOCAL_NAME object, which is thus a new *alias* to the shared object.

The same applies to other kinds of names in the compiler, such as CLASS_NAME or ARGUMENT_NAME for example.

5. ALIASING MUTABLE OBJECTS

Of course, enforcing the use of the aliaser is not enough to safely cope with aliasing. One also has to strictly control who modifies the aliased object and where.

In the context of symbols in the SmallEiffel compiler, an aliased string should never be modified. Doing otherwise is bound to wreak havoc in the compilation process. Thus, it would have been convenient to use some kind of immutable string. However, this has not been possible because Eiffel does not provide such immutable strings. Eiffel STRINGs are indeed very versatile objects: their contents may be changed, extended, shortened, etc. Consequently, some steps have to be taken to verify that the aliased objects are not modified.

Since a library class such as STRING contains many public modifier routines, the best way to control their use would be to encapsulate the STRING object inside, say, an IMMUTABLE_STRING which would feature accessors to the STRING object but no modifier at all, or at least restrictively exported ones. Subclasses of STRING — which currently do not exist in the SmallEiffel compiler context — could also be encapsulated the same way, either by IMMUTABLE_STRING or a subclass of it.

This solution, though very satisfying from a design point of view, has not been applied in SmallEiffel because it would have implied an important penalty both in terms of memory (extra IMMUTABLE_STRING wrappers) and execution time (extra level of indirection).

It is nonetheless possible to take measures to check that the aliased strings are actually not modified. Assertions to this end can be added within the STRING_ALIASER.

As we have explained in section 3.2, the STRING_ALIASER contains an internal, private DICTIONARY named `storage` to store the aliases. In order to guarantee that aliased mutable objects are *not* mutated, a second private DICTIONARY, `immutable_storage`, can be added to the STRING_ALIASER. When a new string is aliased and put in the normal `storage` DICTIONARY, a *copy* (not an alias) of the aliased string is also put into the `immutable_storage` DICTIONARY. Then, each time `item` is called on STRING_ALIASER, an assertion has to be triggered which verifies that the contents of both dictionaries are equal. Since the copies in `immutable_storage` are *never* exposed to the outside of STRING_ALIASER, an assertion failure at this point would show that one of the aliased objects stored in the `storage` DICTIONARY has been modified, thus signaling incorrect behavior from one of the aliases. This is not perfect, however, since copying aliased objects for verification purposes is expensive and since these verifications are performed asynchronously, that is only when `item` is called. Furthermore, it is important to remember that this kind of assertions are effective only when turned on, generally during the development phase. Severe undiagnosed errors may thus occur at run time if the assertions are turned off and the application has not been thoroughly tested.

More generally, aliasing mutable (and actually *mutated*) objects causes less problems when dealing with objects which have been *designed to be aliased*.

An example of such objects are instances of type BASE_CLASS which represent classes after they have been parsed. These objects are much more complex than strings since they include all the information about the class they represent: name, ancestors, attributes, routines, etc. If a class FOO is present in the system, being defined in a `foo.e` file, only one instance of the corresponding BASE_CLASS is ever created, whatever the number of occurrences of FOO in client code. Obviously, aliasing this kind of objects is mandatory. Doing otherwise would only result in a tremendous amount of instances of BASE_CLASS and would be utterly inefficient.

The mechanism used is very similar to the one previously detailed for strings. The alias provider for BASE_CLASS objects is the EIFFEL_PARSER singleton. Its role is to ensure that each class source file is only (lazily) parsed once, for obvious performance reasons, and to provide aliases for BASE_CLASS objects. Indeed, the EIFFEL_PARSER is the only object allowed to create instances of BASE_CLASS. Thanks to the selective export mechanism of Eiffel, this property can be *statically guaranteed*. The following code excerpt shows the export clause for the only creation method of BASE_CLASS:

```
class BASE_CLASS
...
creation {EIFFEL_PARSER} make
...
```

This way, each time a BASE_CLASS object appears in client code, we are sure it is an aliased one coming — directly or not — from the EIFFEL_PARSER. Thus the appropriate checks are performed at compilation time and no extra cost is incurred at runtime, whether in development or in optimized mode. This contrasts with the STRING objects we previously detailed, which may also be created outside the STRING_ALIASER and thus require extra code in the client to ensure their origin at runtime.

This selected export mechanism is also tremendously useful to get a fine-grained control over modifications to BASE_CLASS objects. Unlike for STRING objects, no extra wrapper object is required and no overhead occurs at runtime, be it in terms of memory or speed. Modifier routines (i.e. mutators) can be exported only to the types of objects that need access to them, like this:

```
class BASE_CLASS
...
feature {TYPE}
   smallest_ancestor(type, other: TYPE): TYPE is
...
feature {EIFFEL_PARSER}
   add_index_clause(index_clause: INDEX_CLAUSE) is
```


Prepared using speauth.cls

```
...
feature {RUN_CLASS, PARENT_LIST}
   collect_invariant(rc: RUN_CLASS) is
...
```

It is worth noting that this Eiffel language mechanism — selective export — neatly meets design goals without sacrificing efficiency. We are not aware of any direct equivalent of this in other common OO languages such as C++ or Java.

With BASE_CLASS aliasing it is also very important to make it easier to lazily perform some computations and avoid duplicating them. Looking up the inheritance tree to find the origin of an inherited feature is a very frequent occurrence of such an operation. Since it involves expensive computations which propagate to the ancestors of the BASE_CLASS, it is much better to trigger it only once and then modify the parent list by caching intermediate results for later retrieval. Thanks to aliasing, all occurrences of a given class name in the code share the same BASE_CLASS object, and thus the same aliased list of ancestors. Thus, once a semantic check is performed on one of these occurrences, the corresponding information becomes immediately and inexpensively available to all of them.

One should note that this way, aliases are not notified of mutations in the aliased object. Indeed, there was no need for notification in our application. It is nonetheless extremely easy to add a notification system if needed, thanks to the EIFFEL_PARSER which handles all requests for aliases.

SmallEiffel also features other aliasers based on the same principles, for example for AST nodes, type IDs, customized features [15], etc.

6. EXPERIMENTAL RESULTS

Although a significant amount of literature can be found on aliasing, there seem to be less concrete experimental data quantifying the impact of aliasing from a performance point of view. We provide here this kind of data for aliasing in the SmallEiffel compiler.

As we have shown above, aliasing is very present throughout the SmallEiffel compiler and concerns many kinds of objects. In order to precisely quantify the impact of aliasing, two versions of the compiler should be run: one with full aliasing and another without any. The latter would have meant major and lengthy rewriting of SmallEiffel. Furthermore, as mentioned above, it even seems impossible to write a compiler without some kind of aliasing, e.g. for BASE_CLASS. Consequently, we decided to restrict our benchmarking to string aliasing only.

In order to quantify the impact of string aliasing during the compilation process we thus built a second version of SmallEiffel without string aliasing. It is worth noting this was made easier by the fact that the whole aliasing process was in the STRING_ALIASER. All reference equality tests between two STRINGs were also replaced by the `is_equal` structural comparison.

We benchmarked both versions of the compiler by compiling SmallEiffel itself [§]. This is a significant benchmark because SmallEiffel is a real-world program, fully bootstrapped in Eiffel and represents about 70,000 lines of code for about 300 live classes.

Memory footprint seems to be able to benefit from string aliasing. The total number of strings (words) encountered during the bootstrap is precisely 79,838. All these words contain 637,014 bytes, which amounts to an average word length of about 8 bytes.

Knowing that one STRING object has a memory footprint of 3 words plus the actual contents of the string (the characters), it is easy to compute the total theoretical memory footprint of all unaliased STRING objects on a 32 bits machine:

$$\begin{array}{rclcll} \text{nb\_of\_strings} & \times & (\text{string\_object\_size} & + & \text{average\_content\_length}) & = \\ 79{,}838 & \times & (12 & + & 8) & = \\ & & 1{,}596{,}760 \text{ bytes} & = & 1{,}559 \text{ Kbytes} & \end{array}$$

These 79,838 string occurrences correspond to only 4,265 different strings, whose memory footprint can be estimated at:

$$4,265 \times (12 + 8) = 85,300 \text{ bytes} = 83 \text{ Kbytes}$$

Thus when string aliasing is used, the theoretical memory footprint of STRING objects in the compiler is decreased by 94.6% compared to what it is without string aliasing.

These theoretical figures do not take into account the extra space needed for memory management, such as C `malloc` headers or garbage collector data [16]. Experimental results (figure 1) show that in fact the amount of memory saved by aliasing is 2,456 Kbytes.

Since the whole compiler footprint with aliasing is about 14,708 Kbytes, the benefit *from the sole string aliasing* represents a more than 14% decrease and is thus quite significant. This is even truer considering that SmallEiffel is not only a research project, but also a free product used worldwide by numerous individuals on small personal computers, for which a RAM of 32 Mbytes is not uncommon at all.

This footprint reduction is also a good thing in terms of execution speed, since decreasing memory use also eases the burden on the garbage collector [16] and lowers on-disk swapping.

In fact, the effect of string aliasing on execution time is more than an indirect consequence. Indeed as we wrote in section 2, one of the main reasons for using string aliasing in the SmallEiffel compiler is to be able to do fast string comparisons in client code by merely comparing two references with the `=` operator instead of comparing (the contents of) two objects with the `is_equal` function.

However, the downside of string aliasing is that the management of the STRING_ALIASER has a cost. Indeed, each time a new string is found, it is necessary to check whether it is already in the aliaser and, in case it is not, the new object has to be created and then inserted in the

[§]The results were obtained with a PC Pentium II 333 MHz with 128 Mbytes RAM, running Linux, kernel 2.0.36, and with SmallEiffel -0.78 and egcs-1.1.1.

aliaser. It was thus reasonable to wonder whether the cost of interning strings would outweigh the benefits of mere pointer comparisons.

The experiments we made show (figure 1) that *string aliasing alone* actually leads to an improvement of more than 10% on the whole compilation time.

Once again, it is important to remember that these significant performance improvements come from the sole string aliasing. It is not unreasonable to think that the other kinds of aliasing (on BASE_CLASSes, LOCAL_NAMEs, etc.) also cause footprint and speed improvements which are quite significant. Aliasing under various forms thus seems to play a major part in the very good performance of the SmallEiffel compiler. More performance results pertaining to SmallEiffel can be found in [17, 16, 15].

7. RELATED WORK

In this section we briefly present some of the important work which has been done on aliasing and pointer analysis for quite some time. Since it may cause important problems and debugging nightmares, some of the work related to aliasing has been focused on alias prevention and avoidance, for software engineering purposes. Alias detection and analysis has also been subject to much work in connection with program optimizations techniques.

In [10], a method is described which relies on access modes to limit aliasing and create *islands*. An *island* is a set of objects that can reference each others freely, but for which there is only one access from and to the outside world, in charge of controlling and restricting aliasing. Thus, *intra-island* aliasing is allowed, which makes it possible to use common design idioms relying on aliasing, while *inter-island* aliasing is restricted, making it possible for *islands* to avoid the potential problems caused by aliasing.

Balloons types [18] present some similarities with *islands*. However, *balloons types* are more powerful, being extensions of programming languages that are part of the type system. They allow a more complete control over sharing the state of a data type.

A number of pointer or alias analyses have been studied and algorithms developed. In [19], a context-sensitive pointer analysis relying on Partial Transfer Functions is detailed and shows encouraging results on C programs. However, [20] shows that context-sensitivity does not always improve the analysis and that in the cases they studied context-insensitive alias analysis can be both “surprisingly precise” and efficient.

A technique to avoid pointer-induced aliases is explained in [21], using *unshareable objects* referenced by *unique pointers* which can only be moved but not copied. When an assignment takes place involving unique pointers, the source of the assignment is nullified, thus ensuring that the destination of the assignment will be the only (unique) remaining reference to the object. This allows the implementation of objects which are never subject to aliasing, although at some runtime cost, besides other normal objects which may be aliased.

In [22], alias analysis is performed at low-level, directly on executable code. The major advantage of this approach is that low-level, pointer-related features (pointer arithmetic,...) can be considered, whereas they are not taken into account by higher-, language-level analyses.

Two existing pointer analyses, points-to analysis and connection analysis, are compared in [23]. The paper shows how they can be used to extend and improve well-known optimizations in a compiler, such as loop invariant removal, common sub-expression elimination and array dependence testing.

Flexible alias protection is proposed in [24]. Through the addition of explicit information to each declaration in order to specify the *aliasing mode* of objects, it allows controlling the degree of aliasing and modifications allowed in client code. This technique, which has no runtime cost, makes it possible to avoid problems caused by aliasing while keeping a natural object-oriented programming style.

A formalized type system for (part of) the previous work has been realized in [25]. It relies on *ownership types* to control references visibility and access paths to objects, allowing representation containment.

Java extensions are proposed in [26] to implement *confined types*. The latter are types whose instances are confined in (i.e. cannot be referenced from outside) the package they belong to. *Confined types* incur no runtime overhead and allow catching aliasing and dissemination errors statically, at compile-time.

Three alias analyses are compared in [27]. The first is based on type compatibility information only; the second extends the first by additional high-level field information and the third extends the second with a flow-insensitive analysis. Static and dynamic evaluation show that type compatibility gives cheap but imprecise alias information, which can be significantly improved by the two others. Type-based analysis appears quite effective, and allows extra program optimizations (redundant loads elimination).

A novel inter-procedural, flow- and context-sensitive pointer analysis for multithreaded programs that may concurrently update shared pointers is presented in [28]. This algorithm handles a full range of constructs, is correct and runs in polynomial time. Experimental results show the algorithm has good precision and convergence properties for the tested benchmarks.

In [29], the complications introduced by type casting are first detailed, followed by the presentation of a tunable pointer-analysis framework for handling structures in the presence of casting. Results are given for four implementations of this framework. They show the importance of distinguishing fields of structures and confirm that making conservative assumptions for the sake of portability is rather inexpensive

8. CONCLUSION

As explained previously, our goal in working on aliasing was two-pronged. First, our intention was to use aliasing for optimization purposes. Second, we focussed on software engineering issues, to be able to rely on aliasing in a disciplined way, in order to get the best performance without sacrificing safety.

Although aliasing has significant drawbacks and can sometimes be dangerous, it also offers a certain number of advantages that are useful to developers. We presented our experience, outlining these advantages in the compilation domain and describing the way we were able to use aliasing in implementing SmallEiffel, The GNU Eiffel Compiler.

Since Eiffel, unlike Lisp, Smalltalk or Java, does not support aliasing, either at the language or at the library level, we had to design and implement our own aliasing mechanism for Eiffel. We showed how reasonable design decisions allowed us to do this in a relatively safe way and how some known patterns (i.e. Flyweight and Singleton) coupled to a few idioms of the Eiffel language made safer aliasing easy to implement. In this respect, design by contract, implemented with Eiffel assertions, as well as selective export rules, were powerful and most helpful tools. The Singleton pattern, for which we proposed a novel Eiffel implementation which has now been taken up in [30], has also been crucial. No extension to our implementation language, Eiffel, has been necessary.

The results we measured in terms of memory use and execution time proved to be very positive. By using aliasing on the sole STRING objects, we were able to reduce the memory footprint of these objects twenty-fold. On the whole program footprint, this amounts to a decrease of more than 14%. Similarly, execution speed was improved by more than 10%. These represent a significant part of the overall compilation cost.

The very good performance achieved by SmallEiffel in terms of speed thus seems to owe a lot to aliasing. We believe our experience can benefit other designers and developers, in the wider sense, and not only those coding in Eiffel or developing compilers. Further research work and experience reports are nonetheless needed to more precisely assess the impact of aliasing on various kinds of programs.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful comments and suggestions.

REFERENCES

1. Banning J. An efficient way to find side effects of procedure calls and aliases of variables. *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'79)*. ACM Press, 1979; 29–41.
2. Meyer B. *Object-oriented Software Construction.* Prentice Hall, 1988.
3. Meyer B. *Eiffel, The Language.* Prentice Hall, 1994.
4. Goldberg A. *Smalltalk-80, the Interactive Programming Environment.* Addison-Wesley: Reading, Massachusetts, 1984.
5. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
6. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques and Tools.* Addison-Wesley: Reading, Massachusetts, 1986.
7. McCarthy J, Abrahams PW, Edwards DJ, Hart TP, Levin MI. *LISP 1.5 Programmer's Manual.* MIT Press: Cambridge, Massachusetts, 1962.
8. Goldberg A, Robson D. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley: Reading, Massachusetts, 1983.
9. Gosling J, Joy B, Steele G. *The Java Language Specification.* Addison-Wesley, 1996.
10. Hogg J. Islands: Aliasing Protection in Object-Oriented Languages. *Proceedings of 6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, vol. 26, 1991; 271–285.
11. Hogg J, Lea D, Wills A, deChampeaux D, Holt R. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 1992; **3**(2).
12. Stroustrup B. *The C++ Programming Language.* Addison-Wesley Series in Computer Science, 1986.
13. STL. Standard Template Library Programmer's Guide. http://www.sgi.com/Technology/STL/.
14. ELKS'95. The Eiffel Library Standard, Vintage 95. http://www.gobosoft.com/eiffel/nice/elks95/index.html.
15. Zendra O, Colnet D, Collin S. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. *Proceedings of 12th Annual ACM Conference on Object-Oriented Programming*

Systems, Languages and Applications (OOPSLA'97), vol. 32 of *ACM SIGPLAN Notices*. ACM Press, 1997; 125–141.
16. Colnet D, Coucaud P, Zendra O. Compiler Support to Customize the Mark and Sweep Algorithm. *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, 1998; 154–165.
17. Collin S, Colnet D, Zendra O. Type Inference for Late Binding. The SmallEiffel Compiler. *Joint Modular Languages Conference*, 1997, vol. 1204 of *Lecture Notes in Computer Sciences*. Springer-Verlag, 1997; 67–81.
18. Almeida PS. Ballon Types: Controlling Sharing of State in Data Types. *11th European Conference on Object-Oriented Programming (ECOOP'97)*, vol. 1241 of *Lecture Notes in Computer Sciences*, 1997.
19. Wilson RP, Lam MS. Efficient Context-Sensitive Pointer Analysis for C Programs. *1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, vol. 30. ACM Press, 1995; 1–12.
20. Ruf E. Context-Insensitive Alias Analysis Reconsidered. *1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, vol. 30. ACM Press, 1995; 13–22.
21. Minsky N. Towards Alias-Free Pointers. *10th European Conference on Object-Oriented Programming (ECOOP'96)*, vol. 1098 of *Lecture Notes in Computer Sciences*, 1996.
22. Debray S, Muth R, Weippert M. Alias analysis of Executable Code. *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, 1998; 12–24.
23. Ghiya R, Hendren LJ. Putting Pointer Analysis to Work. *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, 1998; 121–133.
24. Noble J, Vitek J, Potter J. Flexible Alias Protection. *12th European Conference on Object-Oriented Programming (ECOOP'98)*, vol. 1445 of *Lecture Notes in Computer Sciences*, 1998.
25. Clarke DG, Potter JM, Noble J. Ownership Types for Flexible Alias Protection. *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, vol. 33 of *ACM SIGPLAN Notices*. ACM Press, 1998; 48–64.
26. Bokowski B, Vitek J. Confined Types. *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, vol. 33 of *ACM SIGPLAN Notices*. ACM Press, 1999; 82–96.
27. Diwan A, McMkinley KS, Moss JEB. Type-Based Alias Analysis. *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, vol. 33. ACM Press, 1999; 106–117.
28. Rugina R, Rinard M. Pointer Analysis for Multithreaded Programs. *1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, vol. 34. ACM Press, 1999; 77–90.
29. Yong SH, Horwitz S, Reps T. Pointer Analysis for Programs with Structures and Casting. *1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, vol. 34. ACM Press, 1999; 91–103.
30. Jézéquel JM, Train M, Mingins C. *Design Patterns with Contracts*. Addison-Wesley: Reading, Massachusetts, 1999.

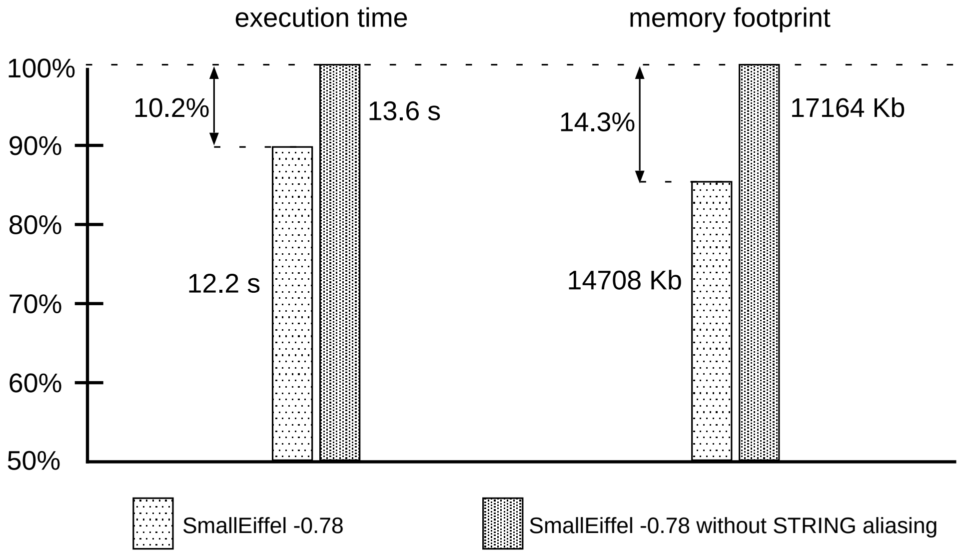

Figure 1. Performance comparison: SmallEiffel with string aliasing vs. SmallEiffel without aliasing