Vers un usage plus sûr de l'aliasing avec Eiffel

Olivier ZENDRA — Dominique COLNET

LORIA
UMR 7503 (INRIA - CNRS - Université Henri Poincaré)
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex, FRANCE

{zendra,colnet}@loria.fr

RÉSUMÉ. *Le code source du compilateur SmallEiffel fait un usage intensif de l'aliasing afin d'atteindre les meilleures performances, tant en termes de mémoire que de vitesse d'exécution. Cette technique semble très appropriée à la compilation mais peut aussi s'appliquer à une large gamme d'applications. Grâce aux capacités de programmation par contrat du langage Eiffel, l'aliasing peut être géré d'une façon assez sûre. Le modèle de conception singleton se révèle également crucial pour l'implantation d'objets fournisseurs d'alias. Nous présentons ici une implantation efficace de ce modèle rendue possible par certains idiomes d'Eiffel.*

ABSTRACT. *The SmallEiffel compiler source code makes intensive use of aliasing in order to achieve very good performance both in terms of memory and execution time. This technique is very appropriate for compilation, but can also be applied to a wide range of applications. Thanks to the design by contract capabilities of the Eiffel language, aliasing can be handled very safely. The singleton pattern also appears to be crucial in implementing alias provider objects. We propose here an efficient implementation of this pattern made easy by some Eiffel idioms.*

MOTS-CLÉS : *alias, partage, Eiffel, sécurité*

KEYWORDS: *alias, Eiffel, safety*

1. Introduction

L'aliasing est connu depuis déjà longtemps (voir par exemple [MEY 88]). L'aliasing est le fait de *référencer une même donnée via deux variables différentes (ou alias)*, ou plus généralement deux chemins différents. La donnée en question peut être aussi simple qu'un booléen, ou aussi compliquée qu'une grande structure de données ou un objet complexe.

On comprend aisément le danger potentiel de l'aliasing puisqu'un développeur qui manipule un objet aliasé n'est pas certain que lui seul contrôle les modifications qui peuvent survenir sur cet objet. Un autre alias peut être utilisé pour changer l'état du même objet. L'aliasing brise donc, en quelque sorte, la séquentialité et l'atomicité apparentes de certains morceaux de code. Ce problème est bien entendu exacerbé lorsqu'on est dans un contexte d'exécution parallèle.

Il est à noter que l'aliasing est la plupart du temps considéré comme un *phénomène* néfaste et devant être évité autant que possible. Beaucoup de travaux se sont donc concentrés sur la détection, la prévention et les moyens d'éviter l'aliasing [HOG 91, HOG 92, MIN 96]. Dans les langages à objets, il semble cependant très difficile — si ce n'est impossible — de l'éviter et des techniques ont donc également été recherchées pour le contrôler.

Nous adoptons dans cet article une approche qui consiste plutôt à considérer l'aliasing comme une *technique* potentiellement utile, offrant certains avantages significatifs en termes de performances. Pour ces raisons, notre compilateur Eiffel [MEY 94], SmallEiffel, The GNU Eiffel Compiler [1], fait un usage intensif mais soigneusement conçu de l'aliasing, essayant de réconcilier performance et sûreté à l'aide de techniques pour la plupart déjà connues.

Cet article, qui rapporte notre expérience avec le compilateur SmallEiffel, est organisé comme suit. Nous expliquons en section 2 pourquoi nous considérons que l'aliasing est très utile dans un compilateur. La section 3 introduit ensuite le concept de *fournisseur d'alias* et détaille comment il peut être mis en oeuvre en Eiffel en utilisant le modèle de conception "singleton" [GAM 94]. Puis nous décrivons dans la section 4 comment la programmation par contrat et les assertions Eiffel peuvent être utilisés pour rendre l'utilisation de l'aliasing plus sûre. La section 5 considère le problème des objets modifiables et les bénéfices dûs à l'utilisation de l'aliasing sont discutés en section 6. Finalement, la section 7 conclut.

2. L'aliasing dans un compilateur: pour quoi faire?

L'aliasing n'est bien entendu pas spécifique à l'implantation des compilateurs. Ceux-ci révèlent néanmoins une forte propension à utiliser l'aliasing. Une des tâches les plus communes d'un compilateur est par exemple de vérifier si une variable est

1. `http://SmallEiffel.loria.fr`

locale, globale, ou est un argument de la routine englobante. Afin d'accomplir cette tâche de façon efficace, il faut faire attention à éviter de coûteuses comparaisons de chaînes par leur contenu. Il est en effet extrêmement fréquent dans un compilateur d'avoir besoin de savoir si deux symboles sont les mêmes ou non.

L'idée de base de l'aliasing est ici d'utiliser exactement la même chaîne de caractères quand plusieurs symboles ont le même nom. Ainsi, il est possible de comparer deux noms de symboles avec une simple comparaison de pointeurs au lieu de devoir examiner le contenu des deux chaînes. Ceci évite aussi les duplicata d'une même chaîne, ce qui économise la mémoire. Il semble également y avoir un gain possible en termes de vitesse. Cependant, le coût de la gestion de l'aliasing peut gommer une partie de ses avantages et doit donc être évalué plus précisément.

Bien entendu, les chaînes de caractères ne sont pas les seuls objets "aliasables" dans un compilateur, mais comme ce sont des objets communs, nous les utiliserons comme exemple de base tout au long de cet article. De nombreux autres types d'objets plus complexes sont aussi de bons candidats à l'aliasing, comme par exemple les différents objets utilisés pour représenter des noeuds de l'arbre abstrait [AHO 77].

3. Utilisation d'un fournisseur d'alias

Les noms de symboles partagés sont apparentés aux "symbols" offerts par Lisp ou Smalltalk, ou aux "interned strings" de Java. Contrairement à ces langages, Eiffel ne propose pas un tel mécanisme, qui doit donc être émulé explicitement par le développeur Eiffel.

Dans SmallEiffel, nous implantons ce type d'aliasing à l'aide d'un *fournisseur d'alias* ou *aliaseur*. En effet, bien que la gestion locale des objets "aliasés" soit très facile et intuitive dans une routine ou un petit algorithme, ce n'est plus le cas lorsque l'aliasing doit être géré à travers l'ensemble du système.

Dans notre compilateur, par exemple, bien que les noms de symboles soient seulement créés dans l'analyseur syntaxique, ils sont accessibles via un objet spécial chargé d'en assurer l'unicité. Cet objet spécial est appelé dans SmallEiffel le STRING_ALIASER. Il présente certaines similarités avec un pont (*bridge*) donnant accès à des îlots (*islands*) d'objets [HOG 91]. Cependant, contrairement à un "pont", notre aliaseur n'a pas pour but de fournir un accès seulement temporaire aux objets qu'il contient (aliasing dynamique [HOG 92]) mais également un accès contrôlé et à long terme (aliasing statique). Pour atteindre son but, il est capital que l'objet STRING_ ALIASER soit unique dans l'ensemble du système. Le modèle de conception "singleton" [GAM 94] est ici parfaitement adéquat.

3.1. *Mise en oeuvre du modèle de conception "singleton" en Eiffel*

Le rôle du modèle de conception "singleton" est de garantir qu'une certaine classe a une instance canonique et de fournir un accès global à celle-ci.

L'implantation C++ [STR 86] proposée dans [GAM 94] garantit que la classe Singleton est instanciée une seule fois grâce à un test explicite lors de l'exécution. En Java [GOS 96] il est possible de garantir sans aucun coût lors de l'exécution qu'une seule instance d'une classe est jamais créée, en rendant le constructeur privé, créant une seule instance statique dans le bloc d'initialisation statique et redéfinissant la méthode `clone` afin qu'elle retourne une référence vers cette instance statique.

En Eiffel, un singleton peut être implanté en se basant sur les mécanismes d'assertions et de routine *once* de la manière suivante:

```
class A_SINGLETON
 -- D'autres primitives ici...
feature {NONE}  -- Partie Singleton
   singleton_memory: A_SINGLETON is
      once
         Result := Current;
      end;
invariant
   is_actually_singleton: Current = singleton_memory
end
```

`Current` est la notation Eiffel pour l'objet courant, comme `self` en Smalltalk [GOL 83] ou `this` en Java ou C++. `singleton_memory` est une fonction Eiffel `once` [MEY 94]. Une telle fonction est exécutée une seule fois, la première fois qu'elle est appelée. Les appels consécutifs retournent exactement le même résultat qui est automatiquement mémorisé pour toutes les instances de la même classe. Comme `singleton_memory` est exportée à NONE, elle est privée et peut être appelée uniquement depuis l'objet `Current`, de type A_SINGLETON ou un de ses sous-types. Ainsi, `singleton_memory` mémorise la référence vers le premier objet de type A_SINGLETON crée.

Les invariants en Eiffel sont des assertions qui sont déclenchées à chaque fois qu'un client appelle une routine sur une instance de la classe englobant l'invariant. Les invariants sont hérités par les descendants des classes où ils apparaissent. L'invariant `is_actually_singleton` garantit qu'un seul objet de type A_SINGLETON (ou sous-types) est jamais créé dans le système. En effet, il vérifie que la référence sur l'objet `Current` est exactement celle stockée par `singleton_memory`. Si plus d'un objet de type A_SINGLETON était créé, cet invariant serait violé et déclencherait une exception car `Current` ferait référence au second objet tandis que `singleton_memory` référence le premier.

Il est important de noter que comme la solution Java mentionnée précédemment, l'implantation Eiffel d'un singleton ne coûte rien en mode optimisé. En effet, les invariants, comme toutes les assertions, sont conçus pour être utilisés durant la phase de développement, mais sont désactivés dans le code final.

Du fait de la sémantique des routines `once` d'Eiffel, chaque classe qui doit être un singleton doit implanter la fonction `singleton_memory` et son invariant `is_actually_singleton`. Hériter de A_SINGLETON n'est pas une solution viable car ainsi il serait impossible d'avoir plus d'un singleton dans le système, le singleton étant commun au type dans lequel il apparaît et à ses sous-types.

3.2. *Implantation d'un fournisseur d'alias*

Grâce à l'implantation des singletons proposée ci-dessus, la spécification de notre fournisseur d'alias devient très simple. En plus du code pour le modèle de conception "singleton", notre STRING_ALIASER offre simplement une fonction, `item`, chargée de fournir une référence vers la chaîne partagée correspondant au modèle spécifié. En voici l'interface:

```
class interface  STRING_ALIASER
   item (model: STRING): STRING
      require
         model /= Void
      ensure
         Result.is_equal(model)
invariant
   is_actually_singleton: Current = singleton_memory
```

La première fois qu'`item` est appelée avec un certain nom comme argument, elle le retourne et mémorise la chaîne `model` dans un DICTIONARY – une sorte de table de hachage – privé. Les appels suivants avec soit la même chaîne modèle soit une chaîne modèle différente de même contenu retourneront tous la référence vers le tout premier objet chaîne mémorisé.

4. Promouvoir et contrôler l'usage du fournisseur d'alias

Si, comme nous l'avons montré précédemment, l'unicité du fournisseur d'alias peut être garantie, cela n'est pas suffisant. Encore faut-il garantir que l'ensemble des objets qui interviennent dans le processus utilisent effectivement ce fournisseur et pas un autre moyen de construction. Ainsi, pour reprendre notre exemple concernant le STRING_ALIASER, il faut impérativement que toutes les chaînes manipulées par le compilateur proviennent de l'objet STRING_ALIASER et qu'aucun intervenant ne diffuse son propre duplicata d'une chaîne "aliasée".

Toutes les classes du système (les intervenants) doivent donc respecter les "règles du jeu" et n'utiliser que le fournisseur d'alias dans leurs interactions avec les autres intervenants. Chacune doit prendre part à la sécurité de l'ensemble du système en vérifiant que ses fournisseurs eux aussi respectent bien la même règle. Grâce aux pré- et postconditions d'Eiffel, un niveau raisonnable de validation peut être atteint.

L'exemple suivant montre comment LOCAL_NAME, la classe qui représente les variable locales, participe à la sécurisation de l'aliasing. La procédure de création `make` prend deux arguments : un pour enregistrer la position dans le code source, l'autre pour l'alias du nom de l'identificateur.

```
class interface LOCAL_NAME  -- Un nom de variable locale.
creation
   make (sp: POSITION; n: STRING)
      require
         sp /= Void;
         n = string_aliaser.item(n)  -- (#1)
      ensure
         start_position = sp;
         to_string = n               -- (#2)
```

Dans l'exemple ci-dessus, la précondition *(#1)* vérifie que l'appelant passe un argument qui est bien une chaîne *aliasée* à l'aide du singleton STRING_ALIASER. En effet si le contenu de `n` est le même que celui d'une chaîne déjà rencontrée et mémorisée par l'aliaseur, ce dernier retourne la référence de la chaîne initialement mémorisée. Si la chaîne `n` n'a pas été obtenue par l'intermédiaire de `string_aliaser`, cette précondition est violée.

Similairement, la postcondition *(#2)* garantit aussi le comportement correct de la classe LOCAL_NAME, c'est à dire que la chaîne aliasée est celle qui est mémorisée dans l'attribut TO_STRING.

Des assertions similaires s'appliquent à tous les intervenants dans le processus de compilation. Par exemple, dans la classe CLASS_NAME qui correspond aux noms de classes ou dans ARGUMENT_NAME pour les noms d'argument formels.

5. Aliasing pour les objets modifiables

Bien entendu, promouvoir l'utilisation du fournisseur d'alias n'est pas suffisant pour travailler en complète sécurité lorsque les objets aliasés sont eux-mêmes modifiables. Les chaînes Eiffel étant des objets modifiables (leur contenu peut être changé, étendu, raccourci, etc.), il faudrait pouvoir s'assurer qu'aucun intervenant dans le processus de compilation ne modifie un alias, qui amènerait sans aucun doute au chaos le plus total dans le processus de compilation.

Puisqu'une classe comme STRING, issue de la bibliothèque standard, contient de nombreuses routines de modification (les mutateurs) publiquement accessibles, la meilleure façon de contrôler leur utilisation serait d'encapsuler l'objet STRING dans, par exemple, une IMMUTABLE_STRING qui comporterait les accesseurs vers l'objet STRING mais aucun modificateur. Cette solution, bien que très satisfaisante du point de vue conceptuel, n'a pas été mise en oeuvre dans SmallEiffel car elle aurait impliqué un coût important tant en terme de mémoire (objets IMMUTABLE_STRING supplémentaires) qu'en vitesse d'exécution (niveau d'indirection supplémentaire).

Il est néanmoins possible de prendre des mesures pour s'assurer que les chaînes aliasées sont en effet non modifiées. Pour ce faire, des assertions peuvent être ajoutées au sein même de STRING_ALIASER. Ceci requiert dans STRING_ALIASER un second DICTIONARY, dans lequel sont conservées des copies (pas des alias) des chaînes aliasées. Puis à chaque fois qu'`item` est appelée sur STRING_ALIASER, une assertion pourrait vérifier que le contenu des deux dictionnaires sont identiques. Comme les copies conservées dans le second dictionnaire ne sont pas accessibles de l'extérieur, une violation d'assertion à ce point indiquerait que l'un des objets aliasés a été modifié, signalant ainsi un comportement incorrect de l'un des intervenant. Ceci n'est cependant pas parfait, car ces vérifications sont réalisées de façon asynchrone, c'est à dire seulement lorsqu'`item` est appelée.

Plus généralement, aliaser des objets modifiables (et effectivement modifiés) pause moins de problèmes lorsqu'on a affaire à des objets qui ont étés *conçus pour être aliasés*. De tels objets sont par exemple les instances du type BASE_CLASS, qui représentent des classes après leur prise en compte par SmallEiffel. Ces objets sont bien plus complexes que des chaînes de caractères car ils contiennent toutes les informations concernant la classe qu'ils représentent: nom, ancêtres, attributs, routines, etc. Si une classe FOO est présente dans le système, définie dans un fichier `foo.e`, une seule instance de BASE_CLASS est créée, quel que soit le nombre d'occurrences de FOO dans le code client. Évidemment, aliaser ce type d'objets complexes est indispensable et il est même difficile d'imaginer le contraire.

Le mécanisme utilisé ici est très semblable à celui précédemment détaillé pour les chaînes de caractères. Le fournisseur d'alias pour les objets de type BASE_CLASS est le singleton EIFFEL_PARSER qui garantit que le fichier source de chaque classe n'est analysé syntaxiquement qu'une seule fois pour d'évidentes raisons de performances. En effet, EIFFEL_PARSER est le seul objet autorisé à créer des instances de BASE_CLASS. Grâce au mécanisme d'exportation sélective d'Eiffel, cette propriété peut être *contrôlée statiquement*. L'extrait de code suivant montre la clause d'exportation de la seule méthode de création de BASE_CLASS :

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

Ainsi, à chaque fois qu'un objet de type BASE_CLASS apparaît dans le code client, on sait statiquement qu'il s'agit d'un objet aliasés fourni — directement ou non — par l'EIFFEL_PARSER. Ainsi les vérifications appropriées sont effectuées à la compilation et aucun surcoût n'apparaît à l'exécution, que ce soit en mode optimisé ou lors du développement. Ceci contraste avec les objets STRING que nous avons précédemment détaillés, qui peuvent aussi être créés sans faire intervenir le singleton STRING_ALIASER.

Le même mécanisme d'exportation sélective est aussi extrêmement utile pour obtenir un contrôle fin sur les modifications des objets BASE_CLASS. Contrairement à ce qui se produit pour les objets STRING, aucune enveloppe supplémentaire n'est nécessaire et aucun surcoût n'apparaît lors de l'exécution, ni au niveau mémoire, ni au niveau vitesse. Les mutateurs peuvent être exportées sélectivement de la façon suivante :

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

Grâce à l'aliasing de BASE_CLASS, il est aisé de réaliser le calcul sur nécessité de la plupart des opérations concernant ces objets. Par exemple, la remontée de l'arbre d'héritage pour trouver l'origine d'une routine héritée est une opération très fréquente qui implique des calculs coûteux se propageant récursivement dans les ancêtres de la BASE_CLASS initiale. Il est beaucoup plus efficace de ne faire ce calcul qu'une seule fois et de mémoriser à l'intérieur de la BASE_CLASS la liste des parents lors du tout premier calcul. Grâce à l'aliasing, toutes les occurrences d'un certain nom de classe dans le code partagent le même objet BASE_CLASS et donc la même liste d'ancêtres mémorisée dans l'unique objet correspondant. Ainsi, une fois qu'un contrôle sémantique est effectué sur l'une de ces occurrences, l'information correspondante devient immédiatement disponible pour tous les autres intervenants.

On peut remarquer que de cette façon, les alias ne reçoivent pas de notifications des modifications de l'objet aliasé. En effet, il n'y avait aucun besoin de notification dans notre application. Dans un contexte autre que celui de la compilation, il serait extrêmement facile de rajouter un système de notification en cas de besoin. Il suffit pour cela de modifier EIFFEL_PARSER qui est l'unique fournisseur d'alias.

SmallEiffel comprend également d'autres fournisseurs d'alias basés sur les mêmes principes, par exemple pour les noeuds de l'arbre abstrait, les identificateurs de types, les routines spécialisées [ZEN 97], etc.

6. Résultats Expérimentaux

Comme nous l'avons montré ci-dessus, l'aliasing est utilisé intensivement dans l'implantation du compilateur SmallEiffel et concerne de nombreux types d'objets. Afin de quantifier précisément l'impact de l'aliasing, les performances de deux versions du compilateur devraient être comparées : une version avec aliasing et une autre sans. Néanmoins, écrire une deuxième version de SmallEiffel sans utiliser l'aliasing n'est pas sérieusement envisageable, voire simplement possible. En conséquence, nous avons dû restreindre nos mesures de performances au seul aliasing des chaînes de caractères. Pour ce faire, nous avons développé une deuxième version de SmallEiffel sans l'aliasing des chaînes. Il est intéressant de noter que cette deuxième version est extrêmement facile à obtenir puisqu'il suffit pratiquement de ne modifier que la classe STRING_ALIASER. Tous les tests d'égalité référentielle entre deux STRINGs ont également été remplacés par la comparaison structurelle faite par `is_equal`.

Nous avons mesuré les performances des deux versions du compilateur lors de la compilation de SmallEiffel lui-même [2] Ce dernier constitue un banc d'essais significatif, car SmallEiffel est une véritable application, complètement auto-compilée en Eiffel et représentant environ 70000 lignes de codes pour 300 classes vivantes.

L'aliasing des chaînes représente une économie mémoire non négligeable. En effet, le nombre total de chaînes (mots) rencontrées durant l'auto-compilation (*bootstrap*) est précisément de 79838. Tous ces mots contiennent 637014 octets, ce qui correspond à une longueur de mot moyenne de 8 octets.

Sachant qu'un objet de type STRING a une occupation mémoire de 3 mots plus le contenu de la chaîne lui-même (les caractères), on peut facilement calculer la taille mémoire totale théorique de tous les objets de type STRING non aliasés sur une machine 32 bits:

$$\begin{array}{rcrcll} \text{nb\_chaînes} & \times & \text{(taille\_objet\_string} & + & \text{longueur\_moyenne)} & = \\ 79838 & \times & (12 & + & 8) & = \\ & & 1596760 \text{ octets} & = & 1559 \text{ KO} & \end{array}$$

Ces 79838 occurrences de chaînes correspondent à seulement 4265 chaînes différentes, dont la taille mémoire peut être estimée à:

$$4265 \times (12 + 8) = 85300 \text{ octets} = 83 \text{ KO}$$

Donc lorsque l'aliasing est utilisé, la taille mémoire théorique des objets de type STRING dans le compilateur est réduite de 94,6 % par rapport à ce qu'elle est sans l'aliasing des chaînes. Ces valeurs théoriques ne prennent pas en compte l'espace supplémentaire nécessaire pour la gestion mémoire interne, comme les entêtes des `malloc` C ou les données pour la gestion du ramasse-miettes [COL 98]. Nos résultats

2. Les résultats ont été obtenus sur un PC Pentium II 333 MHz avec 128 MO de RAM, sous Linux, noyau 2.0.36, et avec SmallEiffel -0.78 et egcs-1.1.1.

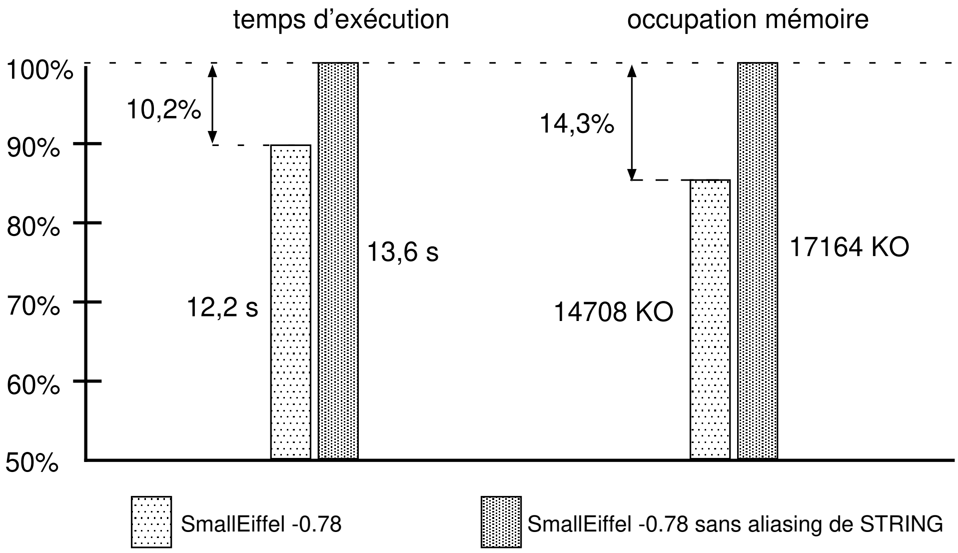

Figure 1. *Comparaison de performances: SmallEiffel avec et sans l'aliasing des chaînes de caractères*

expérimentaux (figure 1) montrent qu'en fait la quantité de mémoire économisée grâce à l'aliasing est de 2456 KO.

Comme la taille mémoire maximale de l'ensemble du compilateur (lorsqu'il compile SmallEiffel) est d'environ 14708 KO, le bénéfice *venant du seul aliasing des chaînes de caractères* représente une diminution de plus de 14 % et est donc très significatif.

Cette réduction d'occupation mémoire est aussi un point positif en termes de vitesse d'exécution, car diminuer la taille mémoire signifie également alléger le fardeau du ramasse-miettes [COL 98] et diminuer les échanges avec le disque (*swapping*).

En fait, les effets de l'aliasing des chaînes sur le temps d'exécution sont bien plus que des conséquences indirectes. En effet, comme nous l'avons écrit dans la section 2, une des raisons principales à l'utilisation de l'aliasing des chaînes dans le compilateur SmallEiffel est de pouvoir faire des comparaisons de chaînes rapides dans le code client, en comparant simplement deux références avec l'opérateur = au lieu de comparer (les contenus de) deux objets avec la fonction `is_equal`.

Cependant, l'inconvénient de l'aliasing des chaînes est que la gestion du STRING_-ALIASER a un coût. En effet, à chaque fois qu'une nouvelle chaîne est rencontrée, il est nécessaire de vérifier si elle est déjà dans l'aliaseur et, dans le cas contraire, le nouvel objet doit être créé puis inséré dans l'aliaseur. Il était donc raisonnable de se demander si le coût pour aliaser les chaînes n'allait pas dépasser les avantages des comparaisons par références. Les expérimentations que nous avons menées montrent (figure 1) que *le seul aliasing des chaînes* conduit en fait à une amélioration de plus de 10 % du temps total de compilation.

Encore une fois, il est important de souligner que cet accroissement sensible des performances vient uniquement de l'aliasing des chaînes de caractères. Il n'est pas déraisonnable de penser que les autres types d'aliasing (sur les BASE_CLASSes, LOCAL_NAMEs, etc.) amènent eux aussi des gains en vitesse qui sont tout à fait considérables. L'aliasing sous diverses formes semble donc jouer un rôle majeur dans les très bonnes performances [COL 97] du compilateur SmallEiffel.

7. Conclusion

Bien que l'aliasing présente des inconvénients importants et soit souvent considéré avec méfiance, il offre également un certain nombre d'avantages qui peuvent être utiles aux développeurs.

Nous avons souligné ces avantages dans le domaine de la compilation et décrit comment nous avons mis en oeuvre l'aliasing dans SmallEiffel, The GNU Eiffel Compiler. Nous avons montré comment des choix de conception raisonnables nous permettaient de faire cela d'une façon plus sûre et comment certains idiomes du langage Eiffel permettent un aliasing plus sûr et plus facile à implanter. Dans ce cadre, la programmation par contrat ainsi que les règles d'exportation sélectives d'Eiffel se révèlent être des outils puissants et utiles. On peut à ce propos remarquer qu'aucune extension du langage n'a été nécessaire. Le modèle de conception “singleton”, pour lequel nous avons proposé une implantation Eiffel novatrice (retenue dans l'ouvrage [JéZ 99]), a également été crucial.

Les résultats en termes d'utilisation mémoire et de temps d'exécution ont été précisément caractérisés et s'avèrent très positifs. En utilisant l'aliasing sur les seuls objets de type STRING, nous avons pu réduire la taille mémoire de ces objets d'un facteur 20. Sur l'ensemble de la taille mémoire du programme, ceci revient à une diminution de plus de 14 %. Similairement, la vitesse d'exécution a été améliorée de plus de 10 %. Ceci est tout à fait important, car il est nécessaire d'avoir, notamment dans le cadre d'environnements de développement incrémental, un processus compilation très rapide permettant des cycles édition-compilation-exécution très courts.

Les très bonnes performances atteintes par SmallEiffel en termes de temps d'exécution semblent donc venir en bonne partie de l'aliasing. Des travaux complémentaires sont néanmoins nécessaires pour pouvoir plus précisément évaluer l'impact de l'aliasing sur différents types de programmes, liés à la compilation ou non.

Remerciements

Nous remercions les relecteurs anonymes pour leurs commentaires et suggestions.

8. Bibliographie

[AHO 77] AHO A. V., ULLMAN J. D., *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977.

[COL 97] COLLIN S., COLNET D., ZENDRA O., « Type Inference for Late Binding. The SmallEiffel Compiler. », *Joint Modular Languages Conference*, vol. 1204 de *Lecture Notes in Computer Sciences*, Springer-Verlag, 1997, p. 67–81.

[COL 98] COLNET D., COUCAUD P., ZENDRA O., « Compiler Support to Customize the Mark and Sweep Algorithm », *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, Octobre 1998, p. 154–165.

[GAM 94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[GOL 83] GOLDBERG A., ROBSON D., *Smalltalk-80, the Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

[GOS 96] GOSLING J., JOY B., STEELE G., *The Java Language Specification*, Addison-Wesley, 1996.

[HOG 91] HOGG J., « Islands: Aliasing Protection in Object-Oriented Languages », *6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, vol. 26, Novembre 1991, p. 271–285.

[HOG 92] HOGG J., LEA D., WILLS A., DECHAMPEAUX D., HOLT R., « The Geneva Convention on the Treatment of Object Aliasing », *OOPS Messenger*, vol. 3, n° 2, 1992.

[JéZ 99] JÉZÉQUEL J.-M., TRAIN M., MINGINS C., *Design Patterns with Contracts*, Addison-Wesley, Reading, Massachusetts, 1999.

[MEY 88] MEYER B., *Object-oriented Software Construction*, Prentice Hall, 1988.

[MEY 94] MEYER B., *Eiffel, The Language*, Prentice Hall, 1994.

[MIN 96] MINSKY N., « Towards Alias-Free Pointers », *European Conference on Object-Oriented Programming (ECOOP'96)*, vol. 1098 de *Lecture Notes in Computer Sciences*, 1996.

[STR 86] STROUSTRUP B., *The C++ Programming Language*, Addison-Wesley Series in Computer Science, 1986.

[ZEN 97] ZENDRA O., COLNET D., COLLIN S., « Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. », *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, vol. 32, Octobre 1997, p. 125–141.