DLP : Développement d'un langage de programmation

Master STL, Sorbonne Université

Année 2022–2023

TME 9 – Cache de méthodes dans le compilateur

Christian Queinnec & Antoine Miné

1 Présentation

Le modèle objet de notre langage utilise une sémantique de type « liaison tardive » pour l'appel de méthode (*late binding* ou *dynamic dispatch* en anglais). Cela signifie que la fonction réellement appelée par l'envoi de message (i.e., l'appel de méthode) `o.m()` est déterminée à l'exécution (i.e., tardivement) et non pas à la compilation. En effet, comme il peut exister plusieurs versions de la méthode `m`, correspondant à des redéfinitions dans des sous-classes, il est nécessaire de rechercher la version correcte. La liaison tardive se base sur la classe avec laquelle l'objet `o` a été créé, ce qui est une information disponible seulement au moment de l'exécution de `o.m()`. Une telle recherche de méthode peut s'avérer coûteuse. Dans la bibliothèque d'exécution du compilateur, cette recherche est effectuée par la fonction `ILP_find_method`.

Le but de ce TME est d'améliorer la performance du code objet compilé par ILP4 en implantant un système de *cache*. Il s'agit d'introduire, à des endroits stratégiques, une mémoire qui se souvient des dernières résolutions de méthodes. Nous stockons, lors d'un appel à `ILP_find_method`, les arguments et le résultat de la fonction. Si, lors du prochain appel à `ILP_find_method`, les arguments sont présents dans le cache, alors le résultat est directement extrait du cache. Sinon, un appel à `ILP_find_method` doit être effectué, et on en profite pour mettre à jour le cache. Ce type d'optimisation a été proposé dès 1984 dans le langage à objets Smalltalk et il reste encore très populaire (en particulier dans les langages très dynamiques, comme JavaScript, où le coût d'une recherche naïve de méthode peut être très élevé).

But :
- comprendre l'implantation de la liaison tardive dans le compilateur ;
- implanter une optimisation de type « cache de méthode » ;
- réaliser et exploiter un banc de test (*benchmark*) pour valider expérimentellement des optimisations.

Ce TME ne concernera que le compilateur, pas l'interprète.
Nous travaillerons dans le *package* `com.paracamplus.ilp4.ilp4tme9`.

2 Travail à réaliser

2.1 Liaison tardive

Il est d'abord nécessaire de bien comprendre le mécanisme actuel de compilation des classes, des objets et des appels de méthodes, pour comprendre les causes possibles de perte d'efficacité. Nous distinguerons en particulier la partie statique, faite une fois pour toutes à la compilation, et la partie dynamique, qui doit être répétée à l'exécution de chaque appel de méthode dans le code C généré. C'est cette dernière qu'il faudra optimiser.

Travail à réaliser : lisez les types `ILP_Class` et `ILP_Method`, et la fonction `ILP_find_method` de la bibliothèque d'exécution. Observez le code C généré sur des exemples utilisant des classes pour comprendre en particulier les structures de données de type `ILP_Class` et `ILP_Method` générées statiquement par la compilation. Répondez aux questions suivantes : quelles opérations sont effectuées par `ILP_find_method` ? Pourquoi ces opérations sont-elles indispensables ? Quel est leur coût ? Quels types de programmes risquent de produire un coût prohibitif ?

2.2 Création d'un banc de test

Le développement d'optimisations de compilateurs ne serait se faire sans une solide évaluation expérimentale. Il arrive fréquemment qu'une optimisation qui semble *a priori* bénéfique soit en réalité contre-productive sur les programmes réels. Même si elle se traduit par un gain, mais si celui-ci n'est pas significatif, il vaut mieux supprimer l'optimisation pour éviter de rendre trop complexe le code du compilateur et risquer d'introduire des bugs.

Travail à réaliser : créez un ensemble de programmes ILP4 qui seront utiles pour tester l'efficacité de l'appel de méthodes. Cette base, initialement réduite, sera amenée à être enrichie dans les questions suivantes, quand le besoin se fera sentir d'évaluer la performance d'optimisations spécifiques. Vous testerez en particulier l'effet de l'héritage avec ou sans redéfinition de méthode, d'une hiérarchie de classes plus ou moins profonde, de séquences d'appels répétés (AAABBB) ou entrelacés (ABABAB), d'appels imbriqués (A appelle B, qui appelle C, etc.). Pour obtenir des mesures de temps significatives, ces programmes effectueront un grand nombre d'appels de méthodes (de l'ordre de 100 000) dans des boucles.

2.3 Cache global de méthodes

Une première idée d'optimisation est de se souvenir, dans des variables globales, des arguments et du résultat du dernier appel à `ILP_find_method`. Lors de l'appel suivant, le résultat en cache est utilisé si les arguments en cache correspondent aux arguments de l'appel. Sinon, l'appel normal est effectué et le cache est mis à jour. Nous considèrons ici, pour simplifier, un cache mémorisant une seule association entre les arguments et le résultat. Si les arguments ne correspondent pas, l'ancienne association est écrasée par la nouvelle.

Pour mesurer l'efficacité de cette solution, outre la mesure du temps d'exécution, on comptera le nombre total d'accès au cache (i.e., le nombre d'appels de méthodes) et le nombre de fois que le résultat souhaité est bien trouvé dans le cache (*cache hit*).

Les arguments d'`ILP_find_method` sont : un objet, une méthode et une arité (nombre d'arguments de la méthode). Le résultat est un pointeur de fonction de type `ILP_general_function`. Toutefois, comme tous les objets de la même classe partagent les mêmes méthodes, il est malin de remplacer, dans le cache, l'information d'objet par l'information de classe. Cela permettra de réutiliser le cache pour un appel de méthode sur un objet différent mais de même classe, augmentant ainsi considérablement son efficacité.

Travail à réaliser : implantez une fonction `ILP_find_method_global_cache` qui ajoute le cache proposé à `ILP_find_method` ; modifiez le compilateur pour appeler cette nouvelle fonction au lieu de `ILP_find_method` ; testez l'absence de régression (et ajoutez ces tests dans l'intégration continue du serveur GitLab du cours).

Testez également la performance en temps d'exécution et en pourcentage de succès de recherche dans le cache. Pour indiquer le gain de performance obtenu grâce au cache, vous pourrez inclure vos mesures dans un fichier texte que vous ajouterez au GitLab du cours, ou bien les indiquer dans l'onglet *Release notes* associé au tag de rendu du projet (vous pouvez copier du texte dans cet onglet ou y attacher un fichier).

Note. Une extension naturelle, que nous ne demandons pas d'implanter ici, serait de stocker plusieurs associations au lieu d'une seule, par exemple avec une table de hachage.

2.4 Cache en-ligne de méthodes

La solution précédente n'est efficace qu'en présence de longues séquences d'appels d'une même méthode, sans entrelacement ni imbrication d'appels, ce qui est un cas peu réaliste. Il est donc nécessaire de changer la stratégie de cache. Une observation empirique est que, si un appel de méthode `o.m(...)` à un point du programme appelle une fonction donnée, alors il y a de grandes chances que, lors des appels de méthodes suivants *au même point de programme*, la même fonction sera appelée. Une telle occurrence syntaxique de l'instruction `o.m(...)` est nommée : *site d'appel*. Ainsi, le programme suivant :

```
1  class Toto {
2    var x;
3    void m() ( print(x) );
4    void n() ( this.m() /* site 1 */ );
5  }
6
7  while ... do (
8    o.m(); /* site 2 */
9    o.n()  /* site 3 */
10 )
```

comporte trois sites d'appel.

L'idée du cache en-ligne consiste à ajouter un cache indépendant à chaque site d'appel. Ce cache se souvient de la dernière fonction appelée *à ce site* et l'associe aux arguments de `ILP_find_method` à ce site, en prévision de la prochaine visite du site d'appel. Puisque les sites d'appels sont générés dans le fichier C par le compilateur, l'ajout du cache nécessitera, entre autres, de modifier le générateur de code C, pour émettre le code supplémentaire de gestion de cache à chaque génération du code d'un appel de méthode.

Travail à réaliser : modifiez le compilateur et la bibliothèque d'exécution pour ajouter un cache en-ligne ; testez l'absence de régression et la performance ; rapportez les mesures de gain de performance dans un fichier texte ou bien dans l'onglet *Release notes* de votre prochain tag GitLab.

2.5 Cache par méthode

L'utilisation d'un cache *en-ligne* a l'inconvénient d'augmenter la taille du programme généré. Une dernière proposition consiste à ajouter un cache par méthode, au lieu d'un cache par site d'appel.

Rappelons qu'une structure de type `ILP_Method` est générée par le compilateur pour chaque nom de méthode déclarée dans le programme. Cette structure est unique étant donné un nom de méthode. En particulier, une même structure sera partagée par toutes les redéfinitions de la méthode dans des classes dérivées. Un cache intégré à `ILP_Method` permettra donc de retrouver rapidement un pointeur vers la fonction appelée dans le cas où la méthode est appelée successivement sur des objets de même classe.

Travail à réaliser : ajoutez un cache de méthode dans la structure `ILP_method` ; testez l'absence de régression et comparez la performance aux autres méthodes proposées plus haut ; rapportez les mesures de performance dans un fichier texte ou dans les *Release notes* de votre prochain tag GitLab.

2.6 Rendu

Vous effectuerez un rendu en vous assurant que tout le code développé a été envoyé sur le serveur GitLab (*push*) dans votre *fork* d'ILP4. Vous vous assurerez que les tests d'intégration continue développés ont bien été configurés et fonctionnent sur le serveur. Vous ajouterez un tag « rendu-initial-tme9 » en fin de séance, puis un tag « rendu-final-tme9 » quand le TME est finalisé. N'oubliez pas d'indiquer les gains de performance obtenus par chaque méthode.