Implémentations Java : compilateur Graal vs. JIT HotSpot

Féaux de Lacroix Martin   Herbert Damien

1 Introduction

"Traditionnellement, il y existe deux approches à la traduction : compilation et interprétation. La compilation traduit d'une langue vers une autre-C vers assembleur, par exemple-avec l'implication que la forme traduite sera plus facile à utiliser pour des exécutions futures, possiblement après d'autres phases de compilation. L'interprétation élimine ces étapes intermédiaires, appliquant les mêmes analyses que la compilation, mais exécutant le programme immédiatement.

JIT compilation est utiliser pour gagner les bénéfices de la compilation (statique) et l'interprétation." - A Brief History of Just-In-Time

Nous citerons donc quelques avantages des deux méthodes et ce que l'on peut en déduire pour ceux du JIT.

1.1 Compilation (AOT)

La compilation (avant l'exécution) effectue les analyses et optimisations avant l'exécution qui prennent un certain temps et peuvent donc ralentir l'exécution si effectuée après.

1.2 Interprétation

L'interprétation peut accéder à des informations durant l'exécution qui sont soit inaccessible avant l'exécution.

1.3 JIT

Comme souligné plus haut, le JIT tente de combiner les avantages des deux méthodes en compilant les méthodes au moment où elles doivent être utilisées, permettant d'effectuer une seule fois l'analyse tout ayant accès aux informations de l'exécution.

2 JAVA et JVM

Java ayant été conçue pour un but industriel, certains choix comme la facilité de distribution des binaires (binaire indépendant de la plateforme, c'est la JVM qui les lits et qui doit être adaptée avec l'hôte) et d'autres comme un important travail pour tenter d'approcher un semblant de sécurité et de performances semble logique. Cependant, tout ce sujet est très vaste et nous concentrons seulement sur le JIT de deux implémentations spécifiques de la JVM : JVM HotSpot et GraalVM.

3 L'éco-système GraalVM

La JVM HotSpot est devenue un monstre de c++ difficilement maintenable et dont la formation à son code peut prendre quelques années. GraalVM est une tentative de pallier ces difficultés créées par la complexité de HotSpot, il est écrit uniquement en Java et remplace une partie de HotSpot dont en

particulier le compilateur C2 avec son compilateur Graal dont nous allons comparer les performances dans la suite.

4 Tired compilation

La JVM utilise le principe de la "tired compilation", c'est-à-dire, un mélange d'interprétation et de JIT, plus une méthode sera utilisée, plus celle-ci sera "affiné" passant par des processus d'optimisation plus poussée, certes plus coûteuses, mais aussi rendant son exécution plus rapide.

Plus précisément, une méthode peut passer par 5 phases différentes :

— la première phase où la méthode est froide, elle est interprétée.
— la seconde phase, la méthode est compilée par C1, en s'appuyant sur les performances mesurées par l'interpréteur
— les phases 3 et 4, la méthode est recompilée en s'appuyant sur l'analyse de performances mesurées de la compilation précédente.
— la phase 5, la méthode est recompilée de manière plus extensive par C2 en s'appuyant toujours sur les analyses précédentes, le code est alors dit chaud.

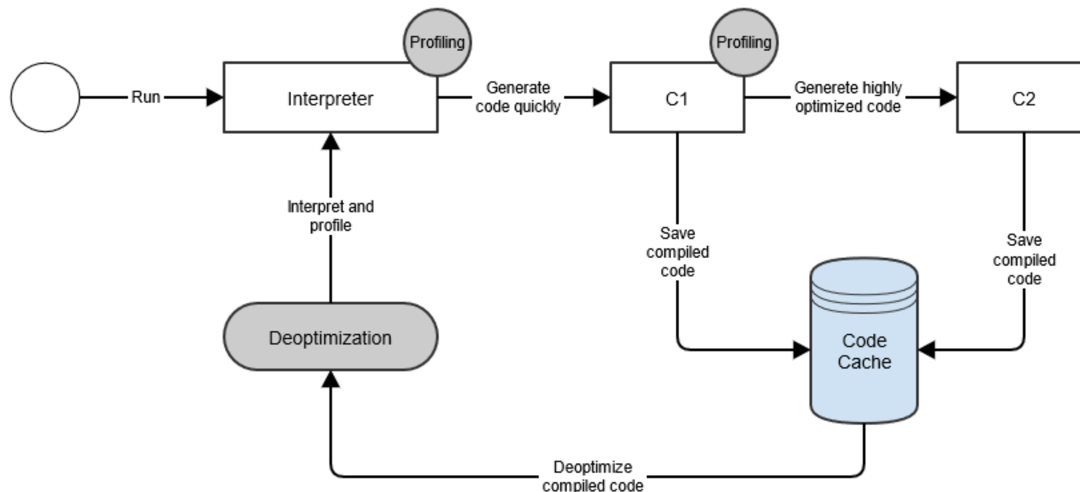

FIGURE 1 – Représentation du processus d'exécution

4.1 C1 compiler

Le compilateur C1, prévu comme compilateur "client", effectuera une analyse locale pour une compilation rapide.

4.2 C2 compiler

Le compilateur C2, prévu comme compilateur "serveur", effectuera une analyse poussée et globale pour optimiser au mieux la méthode avec un coût en temps et en mémoire élevé.

4.3 Graal

Graal se veut plus performant que C2. En effet, il bénéficie de la compilation JIT sur lui-même, ce qui apporte un avantage par rapport à C2 qui est pré-compilé depuis c++. De plus, être écrit en Java lui permet de bénéficier de la gestion de la mémoire interne à la JVM et donc une meilleure répartition de la mémoire entre le code exécuté et le compilateur.

Son écriture en Java est donc LA raison pour laquelle Oracle croit en ce nouveau compilateur et le met en avant depuis plusieurs années.

5 Démonstration

5.1 Mise en place

JVMCI est une interface qui permet de connecter un compilateur Java externe à la place du compilateur C2. Il a été introduit dans la JDK à partir de la version 9. Graal a été incorporé dans la JDK dès la version 10, il se connecte à la JVMCI et peut-être activé via une option expérimentale. Graal a été retiré de la JDK à partir de la version 17 et il est maintenant disponible via GraalVM.

5.2 Le code choisi

Pour la démonstration des performances, nous avons choisi plusieurs agencements de code pour montrer les différentes optimisations possibles par le JIT avec C2 ou Graal. Nous avons aussi ajouté la NativeImage pour montrer que certaines optimisations ne sont disponibles que lors de l'exécution du programme (la NativeImage fait la même optimisation que Graal, mais lors de la compilation au lieu de l'exécution).

5.3 Résultat des performances

Le temps d'exécution d'un programme lancé avec GraalVM, OpenJDK et NativeImage en fonction du nombre de lancements

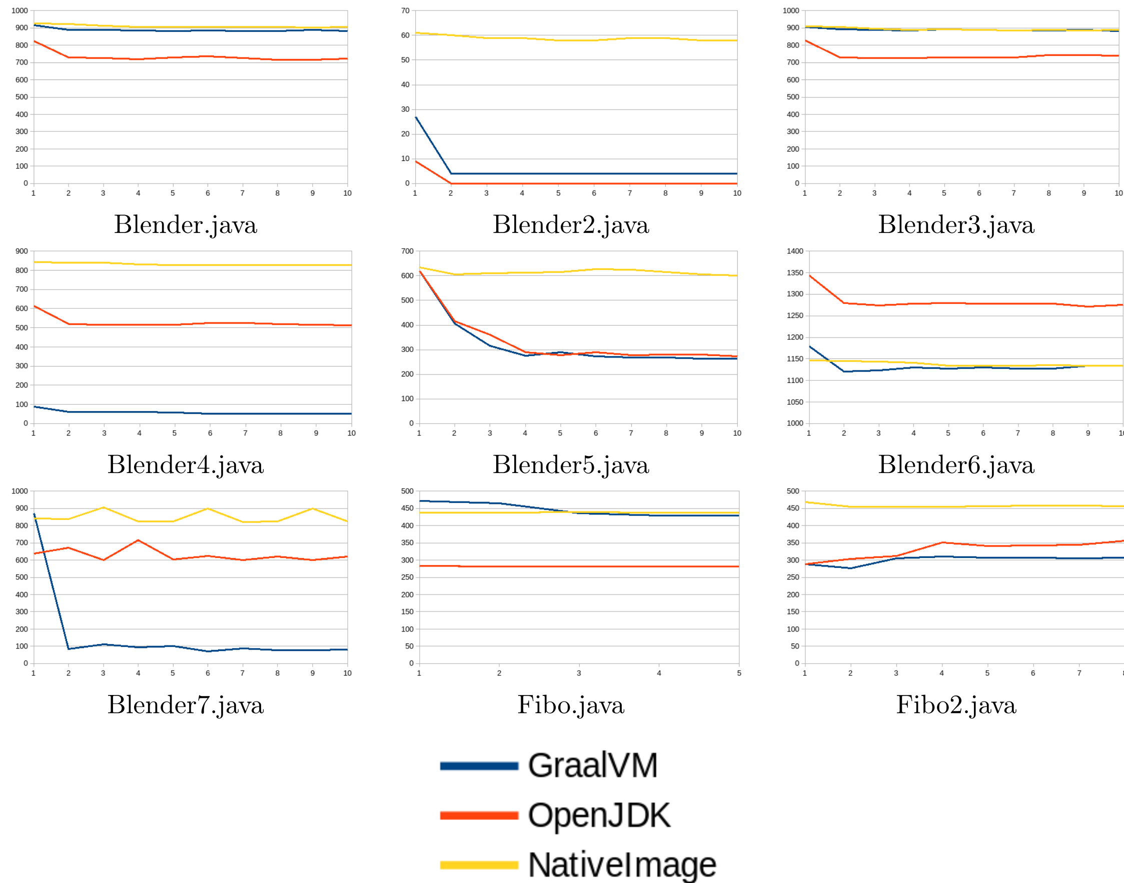

FIGURE 2 – Résultat des performances

En abscisses, il y a le nombre de fois que le code est exécuté, en ordonnée, il y a le temps d'exécution en millisecondes de chaque exécution.

5.4 Analyse des résultats

On peut voir que Graal est plus rapide dans certains cas et que C2 est plus rapide dans d'autres. Par exemple, dans Blender4 et Blender7, Graal est beaucoup plus rapide. Dans tous les cas, Graal est plus rapide que la NativeImage lorsque le code est chaud.

6 Conclusion

Donc le choix entre Graal et C2 dépend de la façon dont votre code est écrit et pour savoir lequel choisir, il faut tester les deux. C'est ce qu'a fait Twitter et ils ont vu que pour leur propre code, ils ont en moyenne entre 5% et 10% de performances en plus avec Graal.

7 Bibliographie

Références

[1] Graal: How to use the new JVM JIT compiler in real life by Chris Thalinger

[2] Baeldung : Tiered compilation

[3] IBM : documentation jit sdk

[4] Documentation oracle : GraalVM compiler

[5] GraalVM documentation : Introduction

[6] HotSpot documentation

[7] Linear Scan Register Allocation - MASSIMILIANO POLETTO and VIVEK SARKAR

[8] Stackoverflow - inlining

[9] Delphine Demange, Yon Fernández de Retana, David Pichardie. Semantic reasoning about the sea of nodes.

[10] A Brief History of Just-In-Time - JOHN AYCOCK

[11] Sparse Conditional Constant Propagation by Mark Anastos

[12] Global code motion/global value numbering - Cliff Click