

Optimisation du compilateur

Projet Génie Logiciel

Fabien Galzi, Abdelmlek Mrad, Lisa Cornu, Matéo Bedel, Eliot Clerc
GL31

Sommaire

Sommaire

Introduction

1. La forme SSA

- a. Analyse bibliographique du sujet
- b. Notre idée initiale
- c. Facilité d'implémentation de SSA après optimisation

2. Elimination du code mort

- A. Introduction
- B. Implémentation
- C. Limites

3. Optimisation d'instructions IMA

- A. Introduction
- B. Multiplication et binaire
- C. Encodage de Booth
- D. Coût en cycles d'horloge interne
- E. Implémentation
- F. Test des multiplications
- G. Le cas des divisions
- H. Test des divisions

4. Propagation de constantes

- d. Introduction
- e. Implémentation
- f. Limites

5. Constant folding

- g. Introduction
- h. Principe de fonctionnement
- i. Implémentation
- j. Limites

Conclusion

2. Annexes

- A. Bibliographie
- B. Capture d'écran du code assembleurs d'optimisations

Introduction

Nous avons choisi comme extension pour notre compilateur, l'optimisation du code généré, l'objectif étant d'améliorer les performances des programmes deca. Notre projet initial était l'implémentation de la forme SSA (Static Single Assignment), une méthode utilisée dans de nombreux compilateurs de nos jours et qui facilite l'implémentation de nombreuses autres optimisations.

Bien que l'implémentation de la forme SSA n'optimise pas directement le code généré, nous souhaitions tout de même l'implémenter sur le conseil de notre professeur référent car le sujet est utilisé dans de nombreux compilateurs modernes pour implémenter ensuite de vrais optimisations et il était donc intéressant de se pencher dessus.

Finalement, nous avons abandonné l'idée d'implémenter la forme SSA pour des raisons qui seront détaillées plus bas, et avons choisi de nous concentrer sur de petites optimisations plus directes, qui seront également détaillées plus loin dans ce document.

Notre documentation a pour objectif de mettre en lumière le processus de réflexion et de conception derrière notre extension. Elle détaillera :

- Notre exploration initiale de la transformation en forme SSA : comment cela fonctionne et ce que nous voulions implémenter
- Pourquoi nous avons finalement renoncé à l'implémenter
- Les principes de conception de nos optimisations ainsi que les choix
- Les stratégies et résultats de validation

Enfin, vous trouverez les sources des documents utilisés pour notre implémentation en annexe.

1. La forme SSA

a. Analyse bibliographique du sujet

La forme SSA est une représentation intermédiaire d'un programme, créée au cours de la compilation et dans laquelle chaque variable est assignée exactement une seule fois. Cette technique transforme la représentation basique du code source en une représentation où chaque assignation d'une même variable reçoit un "identifiant" la rendant unique, qui est généralement représentée par un indice numérique.

La conversion sous la forme SSA implique deux mécanismes principaux : le renommage de chaque variable lors de son assignation et l'introduction des fonctions Phi aux points de convergence de chemins dans le graphe de flot de contrôle, permettant de choisir quelle version de la variable utiliser en fonction du chemin que l'on a parcouru.

Exemple de transformation :

Avant SSA :	Après SSA :
y := 1	y1 := 1
y := 2	y2 := 2
x := y	x1 := y2

Réussir l'implémentation de cette forme nous aurait permis d'implémenter d'autres optimisations bien plus facilement, telles que l'élimination de code mort et la propagation de constante. Nous trouvions intéressant d'implémenter la forme SSA car elle est implémentée par beaucoup de compilateurs modernes, il était très intéressant de se renseigner dessus car nous avons pu nous rendre compte du fonctionnement réel des compilateurs que nous utilisons tous les jours, tel que GCC, et nous regrettons de ne pas avoir eu les moyens de l'implémenter.

b. Notre idée initiale

La SSA (Static Single Assignment) est une forme très intéressante qui permet d'effectuer des optimisations de manière simple et efficace. Cependant, nous nous sommes rendu compte que la mise en œuvre de cette forme pourrait prendre beaucoup de temps, sans garantie de réussite. De plus, elle ne fournirait aucune optimisation directe au départ, alors que notre objectif principal est justement de réaliser des optimisations. C'est pourquoi nous avons décidé de mettre en place directement certaines optimisations que nous aurions implémentées si nous avions utilisé la SSA.

L'idée de départ dans la SSA, comme on le trouve souvent dans les sources, est de diviser le code en blocs, représentés sous forme d'un graph de flux de contrôle (*Control Flow Graph* ou CFG). Cela se fait dans une phase intermédiaire, située entre la génération de code et la vérification contextuelle.

Voici un exemple pour illustrer cette approche.

<pre>Code deca int x=3; Bloc 1 x=4; Bloc 2 if(x==3){ Bloc 3 (la condition) x=79; Bloc 4 } else{ x=5; Bloc 5 } println(x); Bloc 6</pre>	<pre>Bloc : —>5->6 1 ->2 ->3 —>4->6</pre>
--	---

L'idée de l'élimination de variables et du constant folding repose sur la SSA (Static Single Assignment), mais sans l'utilisation des fonctions phi. En effet, bien que la SSA facilite la gestion des variables dans les blocs de contrôle, l'intégration des phi n'était pas évidente

à ce stade. Nous avons donc opté pour une approche différente : au lieu d'utiliser des ϕ , nous avons choisi de stocker les informations dans des tables de hachage, ce qui nous a permis de suivre efficacement l'état des variables sans avoir à manipuler des ϕ . Cependant, il faut noter qu'à ce moment, la gestion des ϕ était complexe et difficile à implémenter de manière fluide dans notre modèle.

Pour être franc, la difficulté principale de la SSA réside dans la manière d'assigner une variable une seule fois. Par exemple, si chaque variable doit être assignée une fois de façon classique, cela peut entraîner une surcharge de la mémoire. On a testé une approche qui consistait à mettre une variable dans un registre, à "bloquer" ce registre, puis à le libérer et à en utiliser un autre lors de la réaffectation. Cependant, cette méthode posait de nombreux problèmes, car il n'y a pas toujours assez de registres disponibles pour gérer cela efficacement. De plus, effectuer des affectations classiques n'a, selon moi, pas d'intérêt, car cela entraîne une consommation excessive de mémoire. Toutefois, l'élimination du code mort pourrait réduire l'espace utilisé, mais cette solution reste loin d'être idéale. C'est pourquoi on a décidé de se concentrer sur les optimisations, plutôt que d'appliquer directement la forme SSA.

c. Facilité d'implémentation de SSA après optimisation

Bien que cela puisse sembler surprenant, après avoir implémenté les optimisations, on se rend compte qu'il y a moins de travail à faire pour mettre en place la forme SSA. En effet, grâce à la propagation des constantes, chaque variable possède désormais une information directe qui permet de la récupérer facilement. De plus, on peut également savoir combien de fois chaque variable a été utilisée.

Cela ouvre la voie à l'ajout d'une structure dans chaque bloc if-else, en plus de la liste d'instructions (qui sont des affectations de variables). Cette structure permettrait de spécifier quelles variables doivent être utilisées. Cependant, la question qui reste à résoudre est la manière de déclarer ces variables une seule fois, sans redondance

2. Elimination du code mort

A. Introduction

L'élimination du code mort est un aspect essentiel de l'optimisation des programmes informatiques. Mais avant de l'aborder, il est important de comprendre ce qu'est un code mort. Il s'agit de parties du code source qui ne sont jamais exécutées ou utilisées, que ce soit des fonctions, des variables ou des blocs de code. Ces éléments inutiles alourdissent le programme et peuvent rendre la maintenance plus complexe. Par exemple, une fonction qui n'est appelée nulle part ou une variable qui n'est jamais utilisée sont des exemples typiques de code mort. L'éliminer permet non seulement de réduire la taille du programme, mais aussi d'améliorer ses performances et sa lisibilité.

Le but est d'arriver à éliminer le code mort afin d'optimiser le programme. Prenons un exemple concret pour mieux illustrer ce concept de code mort.

Avec code mort	Après élimination du code mort
<pre>int x= 1; int y=3; x=3; x=4; x=5; println(x);</pre>	<pre>int x =5; println(x)</pre>

Il existe bien évidemment d'autres exemples, par exemple lorsqu'une méthode effectue un `return` mais que des instructions suivent ce `return`, sans que le programme n'y accède jamais. Cependant, concentrons-nous sur cet exemple précis, car notre optimisation se fera uniquement dans le programme principal.

Je tiens à préciser que l'optimisation est désactivée lorsqu'il s'agit d'objets, et qu'il serait possible d'étendre cette optimisation à une mise à jour future, si cela est souhaité.

B. Implémentation

Pour implémenter cette optimisation, il faut ajouter un mécanisme permettant de suivre l'évolution de chaque variable au fil du temps, afin de déterminer si sa déclaration est réellement utile. La difficulté réside dans la capacité à suivre une variable dans l'arbre, car lorsqu'il s'agit d'objets, l'objet assigné à une variable (par exemple `x`) n'est pas le même que celui défini lors de son initialisation. La seule manière de récupérer cette information est d'utiliser son nom.

Remarque : il est possible qu'il existe une façon plus efficace d'implémenter cette optimisation que ce que nous avons fait, notamment avec la forme SSA par exemple. Mais n'ayant pas pu la mettre en place, nous avons procédé de la manière suivante.

Pour suivre les variables, on utilisera trois tables de hachage. Bien évidemment, chaque table de hachage a pour clé le nom de l'identificateur assigné.

- La première table sert à connaître le nombre d'utilisation d'une même variable (avec comme valeur un entier).
- Cependant, une variable peut être réassignée, et dans ce cas, son nombre d'utilisations peut changer. Pour résoudre cela, il faut une deuxième table, toujours avec le nom comme clé, mais cette fois avec une liste comme valeur. Cette liste contiendra le nombre de fois où l'on a assigné une variable, et chaque élément de cette liste représentera le nombre d'utilisations associées à chaque assignation.

Voici un exemple pour illustrer cette situation :

Code deca	Phase durant la vérification
<pre>int x= 1; int y=3; x=3; x=4; x=5; println(x);</pre>	<pre>table_hash={{“x”,[0,0,0,1]}, {“y”,[0]}}</pre>

En regardant le tableau associé à la variable, on pourra déterminer si cela vaut la peine de produire le code correspondant. Cependant, c'est ici qu'intervient la troisième table de hachage : elle permet d'attribuer à chaque identificateur son bon indice. Pour cela, nous avons ajouté un attribut pour stocker l'indice de la variable.

Lorsqu'il s'agit de récupérer cette information, on fait appel à cette table, où la clé est le nom de la variable et la valeur est un entier correspondant à l'indice actuel.

Dans la génération de code, il suffit de consulter le tableau au bon indice et de vérifier si l'utilisation de la variable est bien égale à 0 pour ne pas produire de code inutile.

C. Limites

Evidemment, cette méthode n'est pas infaillible, et ses limites peuvent être intuitivement liées à la forme normale statique (SSA). En effet, dans la SSA, lorsqu'il y a des branches (`if-else`), le numéro de la variable change en fonction du bloc. Par conséquent, la stratégie précédente ne fonctionne pas dans le cas des structures conditionnelles comme `if-else`, des boucles comme `while`, ou encore des classes. Une mise à jour future pourrait être envisagée pour étendre cette méthode avec une solution adaptée.

3. Optimisation d'instructions IMA

A. Introduction

Dans l'architecture de la machine abstraite IMA, certaines instructions ont des temps d'exécutions (en nombre de cycle d'horloge interne) très inégaux.

Instruction IMA	Nombre de cycle d'horloge interne
MUL	20
QUO	40
SHL	2
SHR	2
ADD	2
SUB	2

On remarque notamment que les multiplications (MUL) et les divisions entières (QUO), sont plus de 10 fois plus lentes que les décalages de bits gauche et droite (SHL et SHR). Or il est possible dans certains cas de remplacer entièrement les multiplications ou divisions par des sommes ou des différences (ADD et SUB) de décalages de bits, en effet décaler n fois les bits d'un nombre par la gauche revient à multiplier ce nombre par 2^n . De même pour les divisions avec un décalage vers la droite.

Multiplications et divisions	Un équivalent en bit shifts
$x*2$	$x<<1$
$x*5$	$x<<2 + x$
$x*113$	$x<<7 - x<<4 + x$
$x/4$	$x>>2$

Comme ces instructions sont courantes, chercher à les optimiser est une idée qui mérite considération. Par exemple, imaginons une boucle while itérant 100 fois sur une multiplication, on effectuera 2 000 au lieu de 20 000 cycles, soit un gain de temps de **90%**.

On cherche donc à optimiser dans le code la multiplication et la division entre une expression quelconque (noté **x**) et un entier (noté **n**).

B. Multiplication et binaire

La première approche à laquelle on pourrait penser pour convertir une multiplication en bit shifts est d'écrire **n** en base binaire et de distribuer le tout. Ensuite on modifie les multiplications de puissance de 2 par des bit shifts et le tour est joué.

Par exemple pour **n = 113** :

$$\begin{aligned} 113x &= (1110001)_2 \times x \\ &= (2^6 + 2^5 + 2^4 + 2^0)x \\ &= (x \ll 6) + (x \ll 5) + (x \ll 4) + x \end{aligned}$$

Cependant cette représentation n'est plus unique quand on rajoute les soustractions, par exemple 113 s'écrit aussi :

$$113x = (x \ll 7) - (x \ll 4) + x$$

Cette représentation est plus optimisée car elle effectue un bit shift et une addition en moins.

C. Encodage de Booth

Pour générer cette représentation on définit une base binaire modifiée où l'apparition de -1 est possible et noté $\bar{1}$.

$$(113)_{10} = 100\bar{1}0001$$

L'encodage de Booth nous introduit cette transformation :

$$\underbrace{1111 \dots 1111}_{k \text{ digits}} \rightarrow 1 \underbrace{0000 \dots 000}_{k-1 \text{ digits}} \bar{1}.$$

On l'utilise ensuite pour retrouver notre représentation : $113x = (x \ll 7) - (x \ll 4) + x$

D. Coût en cycles d'horloge interne

Cependant bien que cette représentation pour $n = 113$, soit efficace dans une architecture plus classique, dans l'architecture IMA elle ne l'est pas.

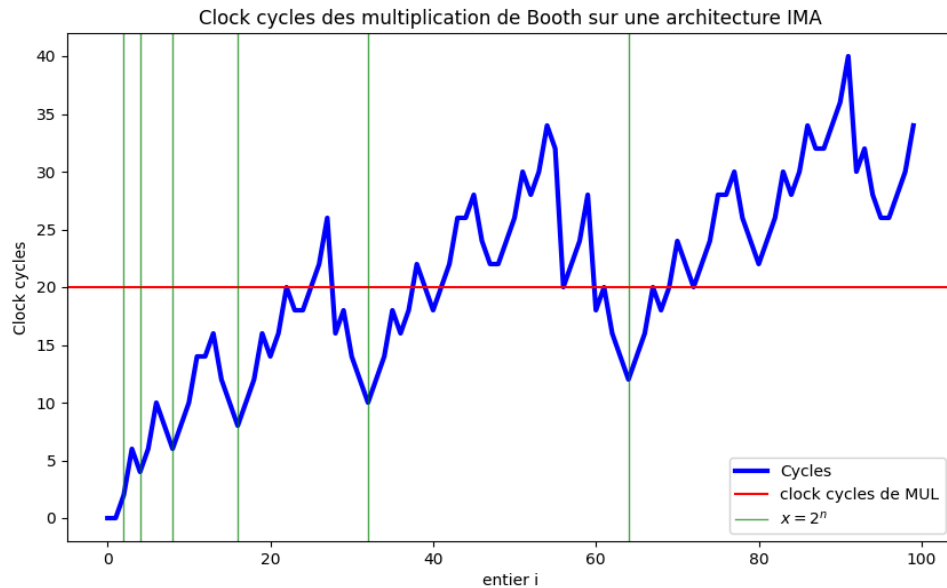
En effet, dans une architecture x86 par exemple, le coût d'un bitshift de i bits par la gauche est constant. Cependant en architecture IMA, on ne dispose que d'une instruction réalisant un bitshift de 1 bit par la gauche (SHL) pour un coup de 2 cycles : décaler de i bits revient donc à réaliser i bitshift, donc a un coût de $2 \times i$ cycles.

Pour $n = 113$, le coût est donc de : $2 \times 7 + 2 \times 4 + 2 + 2 = 26 > 20$

Alors qu'il aurait coûté 8 cycles pour une architecture x86 (si le coût d'un bitshift de i bits est de 2 cycles).

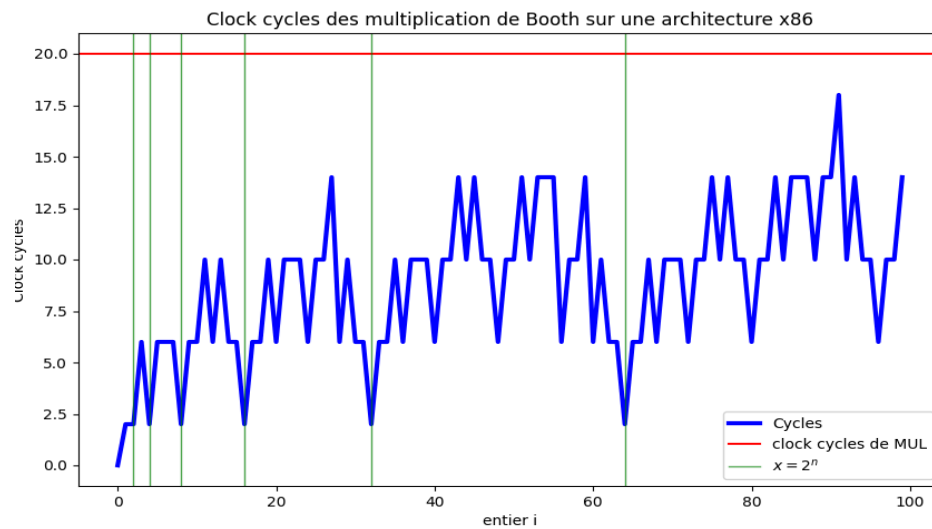
Dans ce cas, il est donc plus rentable de faire une multiplication classique directement. On implémente alors une méthode qui calcule le nombre de cycles qu'effectue cette représentation, et s' il est supérieur à 20, on effectue une multiplication classique pour éviter tout ralentissement.

Cette méthode nous permet aussi d'afficher le coût en cycles d'horloge interne pour chaque entier :



Il est intéressant de remarquer que quand on s'approche d'une puissance de 2, le coût baisse grandement car le nombre de bitshift à réaliser est moins important.

Voici le même graphe que donnerait cette optimisation sur une architecture type x86 :



Mais séchons nos larmes, car cette optimisation reste très efficace pour beaucoup de multiplications quand $n < 100$, qui sont les plus courantes en programmation (rappelons-nous qu'une multiplication par 2 est optimisée à 90% par exemple).

E. Implémentation

Une fois l'algorithme d'encodage de Booth, et la méthode qui vérifie le nombre de cycle d'une multiplication optimisée via cet encodage réalisée (en faisant attention au premier bit de signe), il suffit de coder une méthode récursive qui recrée un sous-arbre de somme ou de différence de bit shifts. Il faudra aussi implémenter de nouveaux nœuds représentant les bit shifts gauches et droits.

On peut ensuite appeler cette méthode quand la multiplication entre un entier et une expression est rencontrée dans la génération de code. Puis on génère le code de ce sous-arbre. L'ensemble des méthodes est implémenté dans une classe *src/java/fr.ensimag/deca/codegen/InstructionOptimiser.class*.

F. Test des multiplications

Pour le programme très simple ci dessous, voici les améliorations observé :

```
{  
    int y = 1;  
    print(0*y);  
    print(2*y);  
    print(3*y);  
    print(20*y);  
    print(64*y);  
    print(66*y);  
    print(113*y);  
    print(128*y);  
}
```

	Multiplication classique	Multiplication rapide de Booth
Nombre d'instructions	72	103
Temps d'exécution	445	361

Bien que le nombre d'instructions augmente (due à l'impressionnant nombre de SHL qui se succèdent), le temps d'exécution diminue significativement, de presque 20%.

G. Le cas des divisions

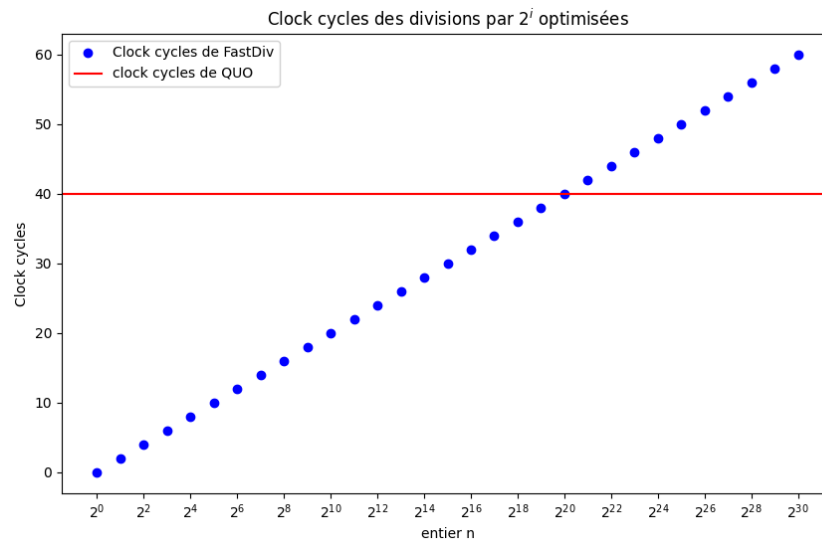
La division est plus contraignante, comme elle n'est pas commutative, on se limitera à traiter le cas x/n .

De plus, sommer des bits shifts ne fonctionne pas aussi bien, puisqu'une fois le dénominateur décomposé on ne peut rien distribuer. On se tiendra donc à optimiser les divisions par des puissances de 2, avec $n = 2^i$.

Par exemple :

$$\frac{x}{1024} = \frac{x}{2^{10}} = x \gg 10$$

Comme pour la multiplication, si n est trop grand, faire des shift bits devient plus lent, bien qu'ici la limite est atteinte bien plus tard (pour $n > 1048576 = 2^{20}$).



On fera donc bien attention, à faire des divisions classiques quand **n** est une puissance de 2 supérieure ou égale à 2^{20} .

H. Test des divisions

Voici un test très simple pour tester le gain de performance (très avantageux pour notre optimisation, reconnaissons-le) :

```
{  
    int y = 1073741824;  
    print(y/1);  
    print(y/2);  
    print(y/3);  
    print(y/4);  
    print(y/256);  
    print(y/4096);  
}
```

De la même façon on observe un temps d'exécution nettement supérieur (de 47% cette fois) :

	Division classique	Division optimisée
Nombre d'instructions	82	86
Temps d'exécution	545	287

4. Propagation de constantes

d. Introduction

La propagation de constantes est une technique d'optimisation utilisée pour améliorer l'efficacité des programmes en remplaçant les expressions constantes par leurs valeurs dès la compilation. L'idée derrière cette optimisation est de simplifier les calculs et de réduire le temps d'exécution en évitant d'évaluer plusieurs fois les mêmes expressions constantes. Par exemple, une opération comme $x + 5$ pourrait être remplacée par une simple affectation si x est également une constante. L'objectif de cette optimisation est de rendre le code plus rapide et plus facile à analyser, ce qui peut avoir un impact significatif sur les performances globales d'un programme.

Prenons un exemple pour illustrer cette optimisation de propagation de constantes.

Avant la propagation des constantes	Après la propagation des constantes
<pre>int x =3; int y =x; int z= x+y; println(z);</pre>	<pre>int x =3; int y =3; int z= 3+3; println(z);</pre>

e. Implémentation

Encore une fois, la difficulté réside dans le fait de récupérer l'information au bon moment, au bon endroit, et pour le bon identificateur. Pour ce faire, pendant la phase d'initialisation, nous allons collecter cette information temporaire en traversant une table de hachage. Cette table contiendra comme clé le nom de la variable et comme valeur un entier (ou un flottant, car dans notre choix, nous avons utilisé deux tables de hachage). Cependant, après réflexion, nous nous sommes rendus compte qu'il serait suffisant de stocker dans une table de hachage de type `Number` en Java, ce qui permettrait de gérer à la fois les entiers et les flottants. Le raisonnement reste cependant

le même.

Ensuite, nous associons un attribut `IntLiteral` (initialisé à `null` pour les entiers, et de même pour les flottants) à chaque variable. Cela nous permettra de récupérer directement l'information sur la variable, et d'utiliser cette valeur lors de la génération du code. Si la variable possède une valeur connue (c'est-à-dire différente de `null`), nous pourrons directement appeler la méthode de génération de code associée à cet attribut.

Pour illustrer cette situation, voici un exemple

Code Deca	Code assembleur avant l'optimisation	Code assembleur après l'optimisation
<code>int x =3;</code> <code>int y =x;</code>	Load #3 , Registre Store Registre ,adresse(x) Load adresse(x) , Registre Store Registre , adresse(y)	Load #3 , Registre Store Registre ,adresse(x) Load #3 , Registre Store Registre , adresse(y)

Grâce à cette méthode, on peut maintenant, au lieu de lire la valeur d'une variable dans la mémoire pour l'assigner, directement récupérer la valeur exacte comme si c'était une constante, et l'assigner à la variable. Cela devient particulièrement utile pour les optimisations suivantes, car une expression comme `x = y + 3` pourra se transformer en `x = 3 + 3`. Je ne vais pas trop en dire pour ne pas gâcher la surprise, mais vous verrez dans la partie suivante à quel point cela peut être puissant pour l'optimisation.

f. Limites

Bien évidemment, cette approche présente des limites similaires à l'autre méthode. En effet, des problèmes peuvent survenir lorsqu'on rencontre des branches ou des boucles. Ainsi, cette optimisation n'est pas appliquée lorsqu'une boucle ou une branche est présente. Pour mieux illustrer cela, lorsqu'il y a des structures de contrôle comme des branches ou des boucles, l'attribut `IntLiteral` (de même pour les flottants) de toutes les

variables sera `null`, ce qui signifie qu'il faut attendre une nouvelle déclaration pour que la variable devienne une constante.

Une autre limite concerne les classes : l'optimisation n'est pas appliquée dans les classes (ou même dans le `main` lorsqu'il y a des classes). L'optimisation est ignorée dans ces cas et ne s'applique qu'aux objets de base.

5. Constant folding

g. Introduction

Le constant folding est une optimisation utilisée dans de nombreux compilateurs qui a pour but d'évaluer la valeur des constantes au cours de la compilation plutôt que de le les évaluer au cours de l'exécution. Cette optimisation a donc pour conséquence de rendre le code généré par le compilateur plus efficace d'un point de vue du nombre d'opérations effectuées mais aussi de le rendre plus court en terme de nombre d'instructions assembleur.

h. Principe de fonctionnement

Nous allons illustrer le principe de fonctionnement du constant folding et donner un aperçu des optimisations qu'il permet à travers le programme Deca suivant :

```
{  
  
    int x = 3 * 6 + 9 % 10;  
  
    println(x);  
  
}
```

En temps normal, sans optimisation un compilateur devrait générer le code assembleur correspondant au calcul de `3 * 6 + 9 % 10` puis le code qui permet de stocker le résultat dans la variable `x` et enfin le code qui affiche ce résultat. Avec le constant folding le résultat du calcul est directement calculé par le compilateur et remplacé par sa valeur

il suffit donc juste de générer le code qui stock ce résultat dans la variable x et le code qui affiche x. Voici les code assembleur produit avec et sans constant folding :

Avec Constant Folding	Sans Constant Folding
<pre> ; Code du programme principal ; Déclaration des variables LOAD #7, R2 STORE R2, 1(GB) ; Instructions LOAD 1(GB), R1 WINT WNL HALT </pre>	<pre> ; Code du programme principal ; Déclaration des variables LOAD #3, R2 LOAD #6, R3 MUL R3, R2 BOV debordement_arithmetique LOAD #9, R3 ADD R3, R2 BOV debordement_arithmetique LOAD #10, R3 CMP #0, R3 BEQ division_zero REM R3, R2 BOV debordement_arithmetique STORE R2, 1(GB) ; Instructions LOAD 1(GB), R1 WINT WNL HALT </pre>

Nous pouvons ainsi observer que le code généré est beaucoup plus concis avec le constant folding car toutes les instructions qui permettent de calculer le résultat de $3 * 6 + 9 \% 10$ ont été retirés et remplacés par le résultat lui-même. A noter que les variables dont il est possible de trouver la valeur à un instant donné de la compilation grâce à la table de hachage évoquée dans la partie traitant de la Constant Propagation sont elles aussi remplacées par leurs valeurs. Par exemple pour le code Deca suivant :

```
{  
  
    int y = 6;  
  
    int x = 3 * y + 9 % 10;  
  
    println(x);  
  
}
```

y va être remplacé par sa valeur donc par 6.

i. Implémentation

L'idée clé de cette optimisation était de pouvoir évaluer le résultat d'un calcul au cours de la compilation. Pour réaliser cela, nous avons écrit une méthode `protected abstract double evalExprValue(DecacCompiler compiler)` de la classe `AbstractOpArith` qui est la classe mère des opérations arithmétique (+, -, %, /, *). Chacune des classe filles override cette méthode, de plus cette méthode a aussi été ajoutée aux `IntLiteral`, `FloatLiteral` et `ConvFloat` pour couvrir tous les calculs possibles. Elle a pour intérêt de retourner le résultat de l'opération effectuée. Ainsi lorsqu'on détecte à la compilation une assignation ou une déclaration de variable dont l'opérande de droite est un `AbstractOpArith`, on charge directement dans le registre concerné par l'assignation du résultat du calcul. Pour les variables qui peuvent apparaître, on utilise la table de hachage décrite dans la partie sur le Constant Folding pour récupérer la valeur de la variable.

j. Limites

Les limites de cette optimisation sont très similaires et liées aux limites du Constant Propagation, en effet les structures `if-else` étant dure à gérer, l'optimisation n'est pas faite, de même pour les classes.

Conclusion

Pour conclure, regardons un petit exemple de nos optimisations, appliquons la compilation au programme deca suivant, ainsi nous obtenons les codes assembleurs suivants, on observe directement que le code optimisé est bien plus succinct.

Code deca	Code Non Optimisé	Code optimisé
<pre> { int a = 1; int b = 2; int c = 3; a = b - 2 + c * 2 - 6 - a + 1; b = c = a; println(a,b,c); } </pre>	<pre> ; ----- Déclaration des variables LOAD #1, R2 STORE R2, 1(GB) LOAD #2, R2 STORE R2, 2(GB) LOAD #3, R2 STORE R2, 3(GB) ; ----- Instructions LOAD #2, R2 LOAD 2(GB), R0 SUB R2, R0 BOV debordement_arithmetique LOAD R0, R2 LOAD #2, R3 LOAD 3(GB), R0 MUL R3, R0 BOV debordement_arithmetique LOAD R0, R3 ADD R3, R2 BOV debordement_arithmetique LOAD #6, R3 SUB R3, R2 BOV debordement_arithmetique SUB 1(GB), R2 BOV debordement_arithmetique LOAD #1, R3 ADD R3, R2 </pre>	<pre> ; ----- Déclaration des variables LOAD #1, R2 STORE R2, 1(GB) LOAD #2, R2 STORE R2, 2(GB) LOAD #3, R2 STORE R2, 3(GB) ; ----- Instructions LOAD #0, R2 STORE R2, 1(GB) LOAD 1(GB), R1 STORE R1, 3(GB) STORE R1, 2(GB) LOAD 1(GB), R1 WINT LOAD 2(GB), R1 WINT LOAD 3(GB), R1 WINT WNL HALT code.Object.equals: TSTO #2 BOV pile_pleine PUSH R2 LOAD -3(LB), R2 CMP -2(LB), R2 SEQ R0 POP R2 RTS </pre>

	BOV debordement_arithmetique STORE R2, 1(GB) LOAD 1(GB), R1 STORE R1, 3(GB) STORE R1, 2(GB) LOAD 1(GB), R1 WINT LOAD 2(GB), R1 WINT LOAD 3(GB), R1 WINT WNL HALT code.Object.equals: TSTO #2 BOV pile_pleine PUSH R2 LOAD -3(LB), R2 CMP -2(LB), R2 SEQ R0 POP R2 RTS	
--	---	--

Nous pouvons aussi constater que le nombre de cycle est bien inférieur pour la version optimisée que pour celle non optimisée :

	Sans Optimisation	Avec Optimisation
Nombre d'instructions	60	48
Temps d'exécution	213	151

En conclusion, pour un code plus long, les optimisations pourraient être très intéressantes. Surtout si ce code a pour objectif d'être utilisé de nombreuses fois.

2. Annexes

A. Bibliographie

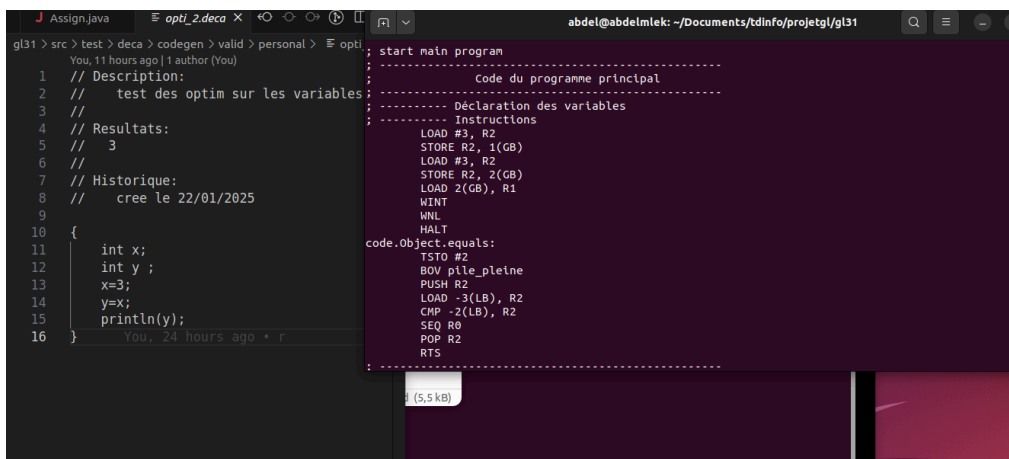
Forme SSA :

1. <https://c9x.me/compile/bib/braun13cc.pdf>
2. https://en.wikipedia.org/wiki/Static_single-assignment_form
3. <https://zestedesavoir.com/tutoriels/427/les-optimisations-des-compilateurs/>
4. https://piazza.com/class_profile/get_resource/hzkg9i9o1ec222/i08j0jat3m63mh
5. https://stefanesco.com/documents/rapport_L3.pdf
6. <http://perso.ens-lyon.fr/paul.feautrier/ssa.pdf>
7. <https://www.geeksforgeeks.org/static-single-assignment-with-relevant-examples/>
8. <https://compilation-course.github.io/ir/4-ssa.html>
9. <https://www.ibm.com/docs/fr/aix/7.3?topic=techniques-compiling-optimization>

Optimisation de la multiplication de Booth :

1. <https://hal-lara.archives-ouvertes.fr/hal-02101792/document>
2. https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

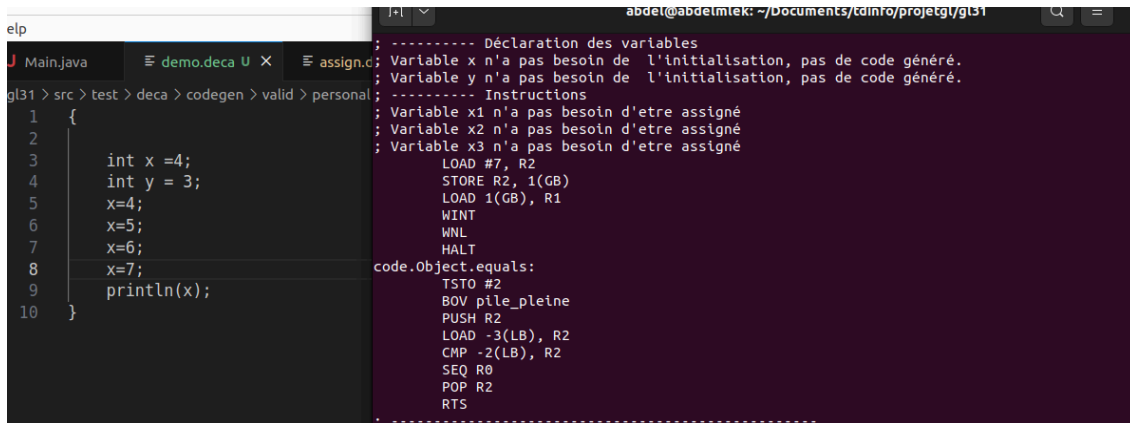
B. Capture d'écran du code assembleurs d'optimisations



```
gl31 > src > test > deca > codegen > valid > personal > opti
You, 11 hours ago | 1 author (You)
1 // Description:
2 // test des optim sur les variables
3 //
4 // Resultats:
5 // 3
6 //
7 // Historique:
8 // cree le 22/01/2025
9
10 {
11     int x;
12     int y ;
13     x=3;
14     y=x;
15     println(y);
16 }
```

```
start main program
-----
Code du programme principal
-----
Déclaration des variables
-----
Instructions
LOAD #3, R2
STORE R2, 1(GB)
LOAD #3, R2
STORE R2, 2(GB)
LOAD 2(GB), R1
WINT
WNL
HALT
code.Object.equals:
TSTO #2
BOV pile_pleine
PUSH R2
LOAD -3(LB), R2
CMP -2(LB), R2
SEQ R0
POP R2
RTS
```

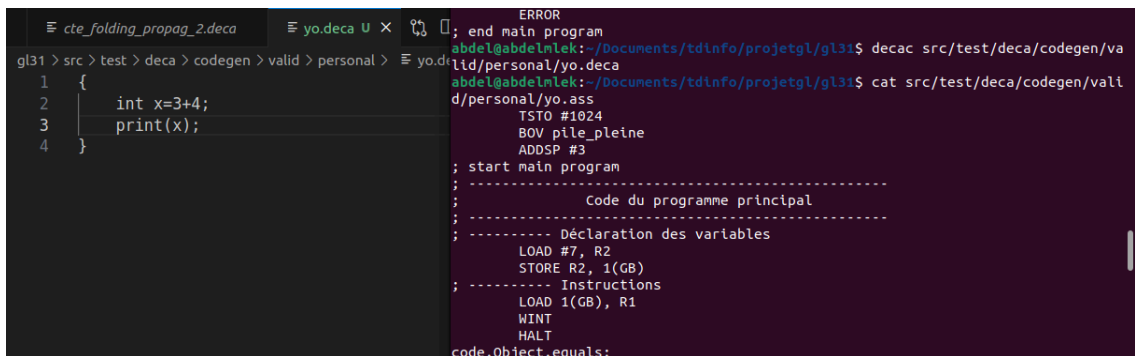

Exemple illustrant la propagation de constantes



```
1 {
2
3     int x =4;
4     int y = 3;
5     x=4;
6     x=5;
7     x=6;
8     x=7;
9     println(x);
10 }
```

```
----- Déclaration des variables
; Variable x n'a pas besoin de l'initialisation, pas de code généré.
; Variable y n'a pas besoin de l'initialisation, pas de code généré.
----- Instructions
; Variable x1 n'a pas besoin d'être assigné
; Variable x2 n'a pas besoin d'être assigné
; Variable x3 n'a pas besoin d'être assigné
LOAD #7, R2
STORE R2, 1(GB)
LOAD 1(GB), R1
WINT
WNL
HALT
code.Object.equals:
TSTO #2
BOV pile_pleine
PUSH R2
LOAD -3(LB), R2
CMP -2(LB), R2
SEQ R0
POP R2
RTS
```

Exemple illustrant l'élimination de code mort



```
1 {
2     int x=3+4;
3     print(x);
4 }
```

```
ERROR
; end main program
abdel@abdelmek:~/Documents/tdinfo/projetgl/gl31$ decac src/test/deca/codegen/val
id/personal/yo.deca
abdel@abdelmek:~/Documents/tdinfo/projetgl/gl31$ cat src/test/deca/codegen/val
id/personal/yo.ass
TSTO #1024
BOV pile_pleine
ADDSP #3
; start main program
; -----
; Code du programme principal
; -----
; Déclaration des variables
LOAD #7, R2
STORE R2, 1(GB)
; ----- Instructions
LOAD 1(GB), R1
WINT
HALT
code.Object.equals:
```

Exemple d'une application du Constant Folding

```

gl31 > src > test > deca > codegen > valid > personal > E ass
Eliot Clerc, 7 hours ago | 2 authors (Lisa Cornu and one other)
1 // Description:
2 // assigne puis affiche des variable
3 //
4 // Resultats:
5 // 2
6 // 5
7 //
8 // Historique:
9 // cree le 10/01/2025
10
11 {
12     int a;
13     int x;
14     int y;
15     x = 2;
16     y = 10;
17     a = 3;
18     a = x + y + a - 10;
19     x = 2;
20     println(x);
21     x = 5;
22     y = 9;
23     x = a;
24     println(x); Eliot Clerc, 10 h
25 }

```

```

BOV pile_pleine
ADDSP #5
; start main program
-----
Code du programme principal
-----
Déclaration des variables
-----
Instructions
LOAD #2, R2
STORE R2, 2(GB)
LOAD #10, R2
STORE R2, 3(GB)
LOAD #3, R2
STORE R2, 1(GB)
LOAD #5, R2
STORE R2, 1(GB)
LOAD #2, R2
STORE R2, 2(GB)
LOAD 2(GB), R1
WINT
WNL
; Variable x3 n'a pas besoin d'etre assigné
; Variable y2 n'a pas besoin d'etre assigné
LOAD #5, R2
STORE R2, 2(GB)
LOAD 2(GB), R1
WINT
WNL
HALT
code.Object.equals:
TSTO #2
BOV pile_pleine
PUSH R2
LOAD -3(LB), R2
CMP -2(LB), R2

```

Exemple combinant les trois optimisations : élimination de code mort, propagation de constantes et propagation de variables