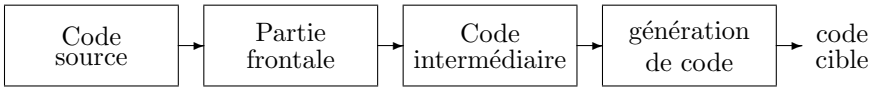
Chapitre 5

Optimisation de code

Il serait souhaitable et intéressant que les compilateurs produisent du code cible de bonne qualité que celui écrit à la main, car il est difficile et pratiquement impossible de le faire.
Avec l'arrivée des nouvelles architectures et des mécanismes au comportement complexe, l'optimisation devient de plus en plus nécessaire, et occupe une place importante dans le processus de compilation. Optimiser un code, ce qui revient à compacter les instructions et à les exécuter de façon rapide, est rarement garanti. L'objectif étant d'en améliorer l'efficacité. Le code obtenu n'est sûrement pas le plus optimal.
Les compilateurs qui effectuent des transformations dans le but d'optimiser sont appelés les optimisants.
Il existe deux types d'optimisation :
- Les transformations de programme qui améliorent le code cible sans prendre en compte les propriétés de la machine cible, telles que :
. La recherche d'un chemin d'exécution le long duquel la valeur d'une variable serait utilisée sans avoir été préalablement initialisée.
. Existence d'un code mort, c'est à dire une partie de programme qui n'est jamais atteinte.
. Existence de variables ayant toujours une même valeur tout le long d'une partie de programme (propagation de constantes).
. Les invariants de boucle, c'est à dire les expressions à l'intérieur d'une boucle qui ne dépendent d'aucune variable susceptible d'être modifiée dans la boucle pour la déplacer hors la boucle.
. Une expression est calculée plusieurs fois sans qu'entre temps, les composantes de l'expression ne soient modifiées (élimination des expressions communes).
. ....

- Les optimisations dépendantes de la machine, telles que :
. L'allocation et assignation de registres.
. l'utilisation de séquences d'instructions spécifiques.
. Exploiter les possibilités de parallélisme du processeur cible.



L'utilisateur peut améliorer le programme (boucles ...)

Le compilateur peut optimiser (les boucles les appels ...)

Le compilateur peut utiliser des registres sélectionner des instructions, transformer, ...

5.1 Transformations conservant les fonctionnalités

Un compilateur peut améliorer un code de différentes manières sans changer sa fonctionnalité. Parmi ces opérations, nous citons : - l'élimination des sous-expressions communes, la propagation des copies, ...

5.1.1 Sous expressions communes

Une occurrence d'une expression E est appelée sous-expression commune, si E a été calculée précédemment et si la valeur des variables apparaissant dans E n 'a pas changé depuis le calcul précédent. Nous pouvons éviter de recalculer l'expression, si nous pouvons utiliser la valeur calculée précédemment.

Exemple

$t_6 := 4 * i$
$x := a[t_6]$
$t_7 := 4 * i$
$t_8 := 4 * j$
$t_9 := a[t_8]$
$a[t_7] := t_9$
$t_{10} := 4 * j$
$a[t_{10}] := x$

Avant optimisation

$t_6 := 4 * i$
$x := a[t_6]$
$t_8 := 4 * j$
$t_9 := a[t_8]$
$a[t_6] := t_9$
$a[t_8] := x$

Après optimisation

5.1.2 Propagation des constantes

Il est fréquent dans un programme de rencontrer une variable dont la valeur est fixée une bonne fois pour toutes. Cette valeur est par la suite utilisée dans des expressions et quelquefois additionnée ou multipliée à d'autres constantes. Il est donc utile de ne pas l'évaluer à l'exécution, mais de le faire à la compilation.

Exemple
$PI := 3.14159$
....
$S := (PI * (D * *2))/4$

peut se reécrire de la façon suivante :
$KI := 3.14159/4$
....
$S := KI * (D * *2)$

5.1.3 Elimination de code inutile

L'analyse d'un programme peut faire apparaître que certains résultats de calcul ne sont jamais utilisés. Par exemple, à partir d'un certain point du programme, une variable n'est plus utilisée, il est donc inutile d'évaluer l'expression qui l'affecte. Il est alors nécessaire de connaître les domaines d'utilisation des variables dans un programme.

Exemple
$x := t_3$
$a[t_2] := t_5$
$a[t_4] := t_3$
transformation si x est inutilisée
$a[t_2] := t_5$
$a[t_4] := t_3$

Remarque
Ce cas peut arriver, sans que le programme l'ai fait explicitement. Cela peut résulter d'une précédente transformation.

5.1.4 Optimisation des boucles

L'existence de boucles dans un programme augmente le temps d'exécution. Ce temps peut être optimisé si le nombre d'instructions dans les boucles est diminué.Trois techniques sont importantes pour l'optimisation des boucles :
- Le déplacement de code
- L'élimination des variables d'induction
- La réduction de force

. Déplacement de code
Il consiste à transférer du code à l'extérieur des boucles.

Exemple
Tant que $i <= limite - 2$ peut être reécrit :
$t := limite - 2$
tant que $i <= t$

. Elimination des variables d'induction
Une variable d'induction est une variable scalaire dans une boucle qui s'incrémente d'une constante à chaque itération. Parfois, on peut éliminer certaines variables d'induction et donc réduire du code. Il faudra seulement veiller à ce que les variables que l'on garde soient initialisées.

Exemple
debut
$j := j - 1$
$t_4 := 4 * j$
$t_5 := a[t_4]$
si $t_5 > v$ aller à début

Nous remarquons que les variables j et t4 sont liées. Chaque fois que j décroît, t4 décroît de 4. De telles variables sont dites d'induction.

On remarque que t_4 est toujours égal à 4*j dans la boucle, on peut alors écrire :

$t_4 = 4 * (j - 1) = 4 * j - 4 = t_4 - 4$

On peut alors réduire le nombre d'instructions.
Le problème qui se pose est que t4 n'a pas de valeur au niveau de cette boucle. On pourra alors écrire :

$t_4 = 4 * j$(nous sortons l'instruction à l'extérieur de la boucle, pour iniatiliser t_4, et gagner du temps en réduisant le code)

dans la boucle :
$j := j - 1$
$t_4 := j - 4$

. Réduction de force (Diminution de la complexité des opérateurs)
Dans certains cas, les multiplications et les exponentiations peuvent être remplacées par des additions ou multiplications (respectivement), car quelque soit le type de machine, les multiplications sont plus coûteuses que les additions (en temps d'exécution) par exemple.

On dit qu'un opérateur OP1 est plus puissant qu'un opérateur OP2, lorsque l'implémentation de OP1 est plus complexe que celle de OP2. Par exemple, l'opérateur de la multiplication est plus puissant que l'opérateur d'addition. L'implémentation câblée de la multiplication est basée sur un circuit logique plus coûteux qu'un additionneur et parfois utilisant plusieurs additionneurs. De même le programme software implémentant la multiplication utilise des opérations d'addition. Le temps d'exécution de l'opérateur OP1 est donc plus important que celui de OP2. Si l'on parvient à substituer un opérateur moins puissant à un opérateur d'une instruction appartenant à une boucle, nous réduirons le temps de calcul de la boucle.

Exemple :
y := 2*x, il vaut mieux écrire y :=x+x
y := x**2, écrire x*x

Remarque : il arrive qu'en réduisant du code, en appliquant les réductions citées dans le cours, on fasse apparaître d'autres réductions.