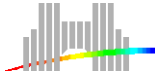
Master Informatique Fondamentale - M1 Compilation

Introduction à l'Optimisation

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

20 mai 2007



Optimisations

- Optimiser les performances
  - Eliminer les calculs inutiles
  - Faire plus vite ou moins souvent les calculs utiles
  - Mieux exploiter la machine cible – parallélisme, hiérarchie de mémoire
- Optimiser la mémoire
  - Programmes
  - Données
- Optimiser la consommation électrique

Toute optimisation repose sur une analyse préliminaire.

Eliminer les calculs inutile

Une instruction est inutile si la valeur qu'elle créé n'est lue par aucune instruction.

- Une instruction est inutile si elle ne figure dans aucune partie "use" d'une "use-def chain"
- Si le progrmme a été mis en forme SSA, une instruction est inutile si la valeur qu'elle est seule a créer n'est jamais utilisée
- Une instruction inutile peut être enlevée, avec des conséquences possibles sur le graphe de contrôle – bloc de base vide, fusion d'arcs, tests inutiles – et sur l'utilité des autres variables.

Algorithme

- Les variables surement utiles sont
  - les paramètres effectifs des procédures (y compris les procédures d'entrée sortie)
  - les arguments des instructions `return`
  - les arguments des tests
- enregistrer ces instructions dans une worklist
- Tant que la worklist n'est pas vide
  - extraire une instruction de la worklist
  - la marquer "utile"
  - ajouter à la worklist toutes les instructions qui définissent des valeurs qu'elle utilise
- les instructions non marquées sont inutiles. Simplifier le graphe de contrôle et recommencer.

Propagation des constantes

- On peut détecter les variables dont la valeur est constante (voir chapitre précédent)
- On remplace une telle variable par sa valeur et on effectue les calculs dans la mesure du possible.
- Il arrive que l'on puisse ensuite éliminer des tests.

Elimination du code mort

Code mort : partie du programme qu'il est impossible d'exécuter.

- Il est rare qu'un programmeur produise du code mort, mais ...
- La propagation des constantes peut engendrer du code mort ...
- Il arrive que des générateurs automatiques, style lex ou yacc engendrent du code mort.
- Algorithme : inondation du graphe de contrôle. Les sommets non marqués sont du code mort.

Invariants de boucle

Le concept de constante est une notion relative.

- Une variable est constante dans une boucle si toutes ses définitions sont à l'extérieur de la boucle
- Une expression est constante dans une boucle si tous ses arguments sont constants
- Une expression (ou une sous-expression) constante peut être calculée une seule fois au dehors de la boucle
- Un test dont le prédicat est constant peut être extrait de la boucle.

```
for(i ....)
  if(p) A;
  else  B;
```

```
if(p)
  for(i ....) A;
else for(i ...) B;
```

Simplification du Contrôle

- Elimination des arcs redondants :

```
  if (p) A; else A;          -->     A;
```

- Elimination des blocs vides

```
    goto A;
    ...                      -->    goto B;
A : goto B;
```

Expressions redondantes

- Analyse des expressions disponibles et affectation à une variable
- Reconnaissance d'une expression disponible, remplacement par la variable
- Elimination des variables inutiles
- Problème des identités remarquables :

```
z = x+y;
...
u = y+x;                    u = z?
```

- Solution : calcul formel.

Réduction de force

Trouver une relation de récurrence entre les valeurs calculées par les itérations successives d'une boucle.

```
for(i=0; i<n; i++)
  k = a * i;
```

- On a la relation de récurrence $k_{i+1} = k_i + a$
- On a remplacé une multiplication par une addition
- Spécialement utile pour les calculs d'indices des tableaux à plusieurs dimensions

Calcul d'indice

```
int a[57][13};

for(i=0; i<n; i++)
  for(j=0; j<n; j++){
    ....
    a[i+j][2*i]
    ...
}
```

- On doit calculer $x = a + 13(i + j) + 4.2.i$
- Récurrence $x_{i,j+1} = x_{i,j} + 13$ et valeur initiale $x_{i,0} = a + 21.i$
- De même, $x_{i+1,0} = x_{i,0} + 21$
- Le calcul peut donc être mené avec un peu plus d'une addition par tour de boucle.

Méthode

- Utiliser l'exécution symbolique pour calculer l'effet du corps de la boucle
- Rechercher les valeurs qui ne dépendent que de l'indice de boucle et de constantes. Soit $e(i)$ une telle valeur
- Rechercher un opérateur $\oplus$ tel que $a = e(i) \oplus e(i-1)$ soit indépendant de i
- A chaque tel opérateur correspond un opérateur *otimes* tel que $e(i) = a \otimes e(i-1)$
- Par exemple, la division permet de remplacer une exponentiation par une multiplication.

Exécution symbolique

- Méthode d'analyse permettant de comprendre le comportement d'un DAG
- Attribuer des noms aux scalaires en début de boucle
- Effectuer les calculs algébriquement
- En un point de jonction, conserver la valeur d'une variable si elle est la même des deux cotés, et lui donner sinon la valeur (absorbante) $\perp$
- Suivre les deux branches d'un test, sauf s'il est possible de décider la valeur du prédicat (peu probable)
- Remplacer la valeur d'un tableau par $\perp$

Optimisation de la localité

L'optimisation la plus "payante" : il peut y avoir un facteur 100 entre le temps d'accés à la mémoire et le temps d'accès au cache.

- Spécialement importante pour les codes scientifiques, car la cache fonctionne mal en présence d'accès réguliers.

```
double a[4000][100];
for(i=0; i<N; i++)
  for(j=0; j<4000; j++)
     a[j][0] = ...;
```

- deux sources d'inefficacité :
  - absence de localité spatiale : `a[j][0]` et `a[j+1][0]` sont à 800 octets de distance
  - absence de localité temporelle : quand `a[j][0]` est accédé pour la deuxième fois, il a été chassé du cache par les autres accès.

Optimisation de la localité, II

Remède : rapprocher le plus possible deux accès au même mot de mémoire, ou à deux mots contigus.
Technique :

- Si possible, réorganiser les tableaux : `a[0][j]` au lieu de `a[j][0]`
- Si possible, réorganiser les boucles : faire tout ce qui concerne `a[j][0]` avant de passer à `a[j+1][0]` :

```
for(j=0; j<4000; j++)
  for(i=0; i<N; i++)
```

- Il existe un critère simple pour savoir si cette transformation (échange de boucles) est légitime.

Parallélisation

On a déjà vu des exemples de parallélisation (ordonnancement d'instructions) On peut aller plus loin :

- On peut calculer les dépendances entre itération d'une même boucle, à condition que les indices des tableaux soient simples :

```
for(i=0; i<n; i++)
  s[i] = s[i-1] + a[i];
```

La valeur créée à l'itération i est utilisée à l'itération $i+1$: il n'y a pas de parallélisme

```
for(i=0; i<n; i++)
  a[i] = b[i] + c[i];
```

Les itérations sont indépendantes, la boucle est parallèle.

- La méthode peut être généralisée : parallélisation automatique.

Réduction de la taille des données

- Intérêt :
  - réduction des défauts de cache
  - réduction du temps de cycle et du coût
- Méthode : analyse des durées de vie. Extension aux tableaux des méthodes utilisées pour les registres

```
ex = 1.0;                                  ex = 1.0;
s[0] = 1.0;                                s = 1.0;
for(i=1; i<=n; i++){                       for(i=1; i<=n; i++){
  s[i] = s[i-1] * x / i;       ==>           s = s * x / i;
  ex += s[i];                                ex += s;
}                                          }
```

car on peut montrer que la variable `s[i]` est morte aprés sa lecture au début de l'itération `i+1`

Réduction de la consommation électrique

- Toutes les optimisations classiques réduisent la consommation :
  - Réduction du nombre de cycles
  - Un accés au cache consomme moins qu'un accés mémoire
- La consommation croit avec la fréquence de fonctionnement du processeur
- Dans un système avec contrainte de performance, toute réduction du nombre de cycle induit une réduction de la fréquence de fonctionnement, donc une économie d'énergie.
- Conjecture : le problème de l'ordonnancement à énergie minimum n'est plus NP-complet.