Génération de Code

L3 Projet Compilation

Paul Feautrier

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

21 février 2009

Première partie I

Questions d'organisation

Les trois phases de la génération de code

La génération du code assembleur ne peut pas se faire en une seule passe :

- Sélection des instructions qui implémentent le code source
- Distribution des données entre la mémoire et les registres (allocation des registres)
- Ordonnancement des instructions

Interactions

Les trois phases ne sont pas indépendantes :

- `x = x + y*z;` :
  - Si le processeur a une instruction MAC : une instruction, pas de registre
  - Sinon, deux instructions et un registre pour loger `y*z`
- `a = b+c;` : le code est différent suivant que `c` est en mémoire ou dans un registre.

Il n'y a pas d'ordre idéal ; recherche de point fixe par itération ?

Heuristiques, I

On ne se préoccupe que des variables originales du programme source. On met de coté suffisamment de registres pour les temporaires nécessaires à l'évaluation des expressions.

- Avantage : le problème est considérablement simplifié.
- Inconvénient : une seule grosse expression peut immobiliser un grand nombre de registres.
- Inconvénient : s'il y a peu de registres, il peut être impossible de loger tous les temporaires.

Heuristiques, II

- On génère le code comme s'il n'y avait pas de registres (ou un seul registre)
- On sélectionne les variables les plus utilisées et on les loge en registre
- On corrige le code.
- Avantage : méthode idéale pour un processeur ayant peu de registres
- Inconvénient : lorsqu'une variable n'est plus utilisée, il est difficile de récupérer son registre.

Heuristiques, III

- On génère le code comme s'il y avait une infinité de registres (registres virtuels)
- On tente d'affecter chaque registre virtuel à une registre réél, en tenant compte des durées de vies des données.
- Si c'est impossible, on ajuste le code en éjectant quelques informations en mémoire.
- Méthode idéale si la machine a beaucoup de registre. Empiriquement, il est rare d'avoir besoin de plus d'une vingtaine de registres.

Le problème des alias

Il y a *alias* quand une même cellule de mémoire peut être atteinte de plusieurs façons différentes.

```
int x; int *p;
...
x = 0;
...
p = &x;
...
*p = 1;
```

- `*p` est un alias de `x`
- Il est impossible de loger `x` dans un registre
- Il est impossible de remplacer `x` par 0

Alias, exemples

- Dans tous les langages, on peut créer des alias par transmission de paramètres par référence :

```
float foo(float A[], float B[]){ ... }

float X[100];
foo(X,X);
```

- En Fortran, il y une instruction *ad hoc* : `EQUIVALENCE`

Analyse d'alias

- Il est possible (mais difficile) de rechercher les alias dans un programme
- On est souvent obligé de se contenter d'une analyse pessimiste
- Dans certains langages, l'analyse est simplifiée : exemple de Java
  - Une variable locale ne peut pas être un alias d'un membre de classe
  - Deux membres de classes différentes ne peuvent être en alias
  - Deux références ne peuvent être rendues égales que par une affectation explicite.

Conclusion provisoire

Point de départ de la génération de code :

- Une représentation interne linéarisée
- On a identifié les variables qui doivent rester en mémoire (variables volatiles)
- En général, on a développé l'adressage :
  - Indexation
  - Accès aux variables non locales
- On commence par sélectionner les instructions
- On alloue les registres et on corrige le code
- On ordonnance les instructions

Deuxième partie II

Sélection d'instructions

Présentation du problème

Les instructions d'affectation des langages de haut niveau n'ont pas de correspondant direct en langage machine. Elles doivent être décomposées en instructions plus simples.

```
QBE(LX0+LOCT)=TF*CBE+PE*CZBE*(1.0D0-ARG*SARG)/(1.0D0-XME)
```

Un extrait du benchmark SPICE, en Fortran

```
F1 = TF*CBE;
F2 = 1.0D0 - XME;
F3 = ARG * SARG:
F3 = 1.0D0 - F3;
F2 = F3 / F2;
F2 = F2 * PR;
```

```
F2 = F2 * CZBE;
F1 = F1 + F2;
R1 = &QBE;
R1 = R1 + LX0;
R1 = R1 + LOCT;
*R1 = F1;
```

Complexité du langage machine

En général, il y a plusieurs façons d'obtenir le même résultat en langage machine.

- Remettre à zéro un registre :
- `LOAD R1,zero`
- `LOADI R1, 0`
- `XOR R1,R1,R1`

C'est en général la troisième méthode la plus rapide.

Complexité de l'algèbre

Les identités remarquables de l'algèbre (essentiellement la distributivité) compliquent la génération de code.

- Il vaut mieux calculer $(a+b)^2$ que $a^2+2ab+b^2$ (une addition et une multiplication contre 3 multiplications et deux additions)
- Quelle est la meilleure façon de calculer ax^3+bx^2+cx+d ?
  - Schéma de Horner : $((ax+b)x+c)x+d$. OK mais pas de parallélisme.
  - $(ax+b)x^2+(cx+d)$ Une multiplication de plus, mais parallélisme.

RISC

Le plus simple.

- Un grand nombre de registres.
- Tous les calculs se font entre 3 registres

```
R1 = R2 + R3;
```

- Une instruction de lecture `LOAD` et une instruction d'écriture `STORE`.
  - Calcul d'adresse : un registre de base, plus un déplacement
  - Parfois, un registre index :

```
LOAD R1,R2,R3,disp  ====>  R1 = *(R2 + R3 + disp);
```

CISC, I

- Grand nombre d'instructions. Exemple : Intel IA32 : plus de 230 instructions. Les compilateurs n'en utilisent qu'un sous-ensemble.
- Chaque instruction est du type 2 adresses et détruit l'un de ses opérandes :

```
R1 = R1 + R2;
```

- Presque toutes les instructions peuvent prendre un opérande en mémoire.
- Il existe de très nombreux modes d'adressage (en général un registre plus un déplacement) : plus de 20 modes pour IA32.

Utilisation des registres

- Le nombre de registres est limité (héritage).
- Il peut y avoir des contraintes ...
- Un nombre en double précision occupe deux registres de numéros pair et impair
- ... et des recouvrements.
- En IA32 AH . AL = AX : deux registres 8 bits composent un registre 16 bits, lui-même partie d'un registre 32 bits, EAX.

CISC, II

Il existe parfois des instructions de haut niveau qui ne correspondent pas à des instructions du langage source. Exemples :

- Multiply and accumulate en une seule instruction :

```
R1 = R2 * X + R3;
```

- Opération de copie avec répétition :

```
while(ECX--) *EDI++ = *ESI++;
```

- Opérations SIMD ou "multimedia" :

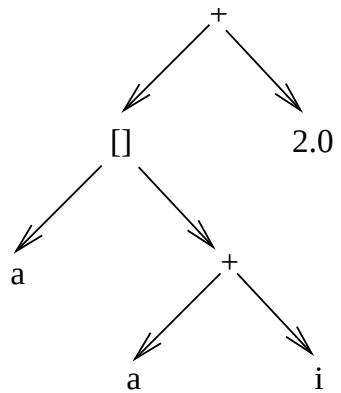
```
char x[], y[], z[];
for(i=0; i<8; i++) x[i]=y[i]+z[i];
```

La méthode naïve, I

Structure de données :

- La génération se fait instruction par instruction.
- Une représenation arborescente d'une instruction d'affectation.
- Un nœud = une opération, unaire ou binaire.
- Une feuille = une variable ou une constante.
- Il faut développer les indexations :

```
a[i+1] + 2.0
```



La méthode naïve, II

Registres virtuels :

- Si l'instruction est trop complexe, il faut la calculer en plusieurs étapes et noter les résulats dans des temporaires.
- On utilise pour cela des *registres virtuels* que l'on suppose en nombre illimité. On allouera les registres virtuels aux registres physiques dans une phase ultérieure.

Emission des instructions assembleur :

- La méthode la plus simple est d'ouvrir un fichier et d'écrire le code assembleur au vol.
- Inconvénient : la post-optimisation devient impossible.
- On peut aussi définir une représentation intermédiaire :

```
type operation = {opcode : string;
                  r1, r2, r3 : int}
and load = {base, index : int;
            displacement : string}
and ...
and intruction = Op of Operation | Load of load | ...
and program = instruction list
;;
```

Génération par descente récursive, I

```
Codegen(p)  switch type de p do
    case opérateur binaire
        r1 := Codegen(p.filsGauche);
        r2 := Codegen(p.filsDroit);
        r0 := nouveau registre virtuel;
        emettre OP r0,r1,r2;
        libérer r1, r2 ; return r0;
    end
    case nombre entier
        r0 := nouveau registre virtuel;
        emettre LOADI r0,p.valeur;
        return (r0);
    end
end
```

Génération par descente récursive, II

```
Codegen(p)  switch type de p do
    ...;
    case identificateur
        r0 := nouveau registre ;
        emettre LOAD r0,base,p.déplacement;
        return (r0);
    end
    case affectation
        r1 := Codegen(p.filsDroit);
        emettre STORE r1,base,p.deplacement;
        return (r1);
        [ libérer r1 ; ]
    end
    ...
end
```

Critique

- Rigidité : une même opération est toujours compilée de la même façon.
- A chaque instant, l'algorithne ne "voit" qu'un seul nœud de l'AST.
- Il est difficile d'exploiter les optimisations non locales :
  - Sous-expressions communes
  - Valeurs déja présentes dans un registre

Génération de code par réécriture d'arbre, I

Une règle de réécriture se compose de trois parties :

- Un *redex*, qui est un arbre pouvant contenir des variables
- Une *réduite*, idem. Une variables peut apparaître à la fois dans le redex et dans la réduite. Une variable qui n'apparaît que dans la réduite et traitée spécialement, voir plus loin.
- Un *modèle* : une chaîne de caractères représentant une instruction assembleur, dans laquelle peuvent apparaître les variables du redex et de la réduite.

Dans ce qui suit, on représente les arbres en notation préfixe parenthésée. Les variables sont des lettres grecques.

Exemple

La règle qui permet de reconnaître les MACs :

- le redex :

$$:= (R(\alpha), +(R(\alpha), *(R(\beta), R(\gamma))))$$

- la réduite :

$$R(\alpha)$$

- le modèle :

$$\texttt{MAC R}\alpha\texttt{,R}\beta\texttt{,R}\gamma$$

- Interprétation : il est équivalent de calculer le redex ou bien d'exécuter le modèle puis de calculer la réduite.

Génération de code par réécriture d'arbre, II

Algorithme du générateur :

- Trouver dans l'AST un sous-arbre isomorphe à l'un des redex. Noter la nécessaire substitution.
- Générer des valeurs pour les variables de la réduite qui ne figurent pas dans le redex.
- Enlever le redex et greffer la réduite à la place, après substitution.
- Emettre le modèle après substitution.
- Continuer aussi longtemps que possible.

Génération de code par réécriture d'arbre, III

Construction de la table des règles

- Chaque instruction *utile* du langage machine doit avoir sa règle
- Chaque opérateur du langage source doit avoir sa règle.
- Chaque opérateur peut avoir plusieurs règles :

$$
\begin{aligned}
/(R(\alpha), I(2)) &\rightarrow R(\beta) : \texttt{MOV}\ \beta, \alpha; \texttt{SHIFTR}\ \beta, 1 \\
/(R(\alpha), R(\beta)) &\rightarrow R(\gamma) : \texttt{DIV}\ \gamma, \alpha, \beta
\end{aligned}
$$

Complexité, optimisation

En gros, pour trouver le point d'application de la bonne règle, il faut un temps $O(R.N)$: R nombre de règles, N taille de l'AST.
Pour optimiser :

- Ranger les règles dans une table hashcodée sur l'opérateur de tête.
- Quand plusieurs règles sont possibles, les essayer dans l'ordre d'efficacité décroissante.
- Elaborer des heuristiques de déplacement dans l'AST.
  - Commencer par les feuilles de l'AST
  - Quand on a réduit un sous-arbre, essayer de réduire ses frères, puis son père.

Bilan

- La méthode permet de construire des compilateurs portables : il suffit de redéfinir le système de régles.
- Il est possible d'y incorporer certaines optimisations (division par une puissance de 2, recherche de sous expressions communes).
- La complexité est proportionelle au nombre de règles, mais reste raisonable.