Compilation

TD 7 : Génération de code

G. Iooss & C. Alias

Précédemment, nous avons vu comment mettre le programme sous forme d'une représentation intermédiaire (*Control Flow Graph*) afin de manipuler (ex : en passant sous *forme SSA*) et d'optimiser le programme à loisir (ex : détecter et supprimer les calculs redondants).

Cependant, le plus gros reste à faire, à savoir... la génération du code. Plusieurs questions surgissent, même dans le cas d'un unique basic-block :

- (**Sélection d'instruction**) Quelles instructions doit-on utiliser pour le calcul ?
- (**Ordonnancement**) Quand/Dans quel ordre doit-on les exécuter ?
- (**Allocation de registre**) Où/Quelles ressources doit-on utiliser ?

Partie I - Sélection d'instruction

Exercice 1. Sélection d'instruction

Considérons un langage assembleur ayant le jeu d'instruction suivant :

- `add_const` (coût : **1**)
- `mult_const` (coût : **2**)
- `add` (coût : **1**)
- `mult` (coût : **2**)
- `ldi` (coût : **1**)
- `mult_add` (coût : **3**)
- (les accès aux registres ont un coût de **0**)

Questions :

- Appliquez l'algorithme vu en cours pour trouver le coût minimal d'une suite d'instructions calculant l'expression $(4 + (2 * 3)) * (6 * a)$, a étant déjà stocké dans un registre.
- Faites de même sur l'expression $(4 + (2 * 3)) + (2 * 3)$. Faut-il mieux dupliquer ?
- On suppose que le `mult_add` ne coûte plus que **1**. Maintenant, faut-il mieux dupliquer ?

Partie II - Ordonnancement

Exercice 2. Algorithme de Sethi-Ulmann et ordonnancement

Considérez le code de basic-block suivant :

```
delta = b1 * b2 - 4*a*c
```

On suppose que les variables `b1`, `b2`, `a` et `c` sont stockées sur la pile aux adresses `[ARP-0]` à `[ARP-3]`. De plus, on suppose que `delta` ira à l'adresse `[ARP-4]`.

Questions :

- Donnez un code assembleur qui calcule `delta` avant de le remettre sur la pile.
- [Sethi-Ulmann] Construisez le DAG du code précédent.
- En appliquant l'algorithme de Sethi-Ulmann, trouvez le nombre minimum de registres devant être utilisé. Donnez le code assembleur réordonné correspondant.
- [Prefetching] Donnez le code assembleur réordonné de telle façon à préfetcher toutes les variables stockées sur la pile dans les toutes premières instructions.
- Calculez les durées de vie. Quel est le nombre de registres minimum devant être utilisés ? En supposant que l'on a uniquement 3 registres, où faut-il spiller ?
- Trouvez à la main un code n'utilisant que 3 registres, préfetchant les données le plus tôt possible, sans spiller aucune variables.

Exercice 3. Optimisation globale

Considérons une architecture (fictive) où un accès mémoire (`load` ou `store`) prend **4 cycles** tandis que n'importe quelle autre instruction en prend qu'**un seul**. Considérons le programme suivant :

```
for (int i=0; i<6; i++)
    V[i] = 2*A[i];
```

Questions :

- En appliquant les méthodes vu en cours, construisez le CFG de ce programme et déduisez le code assembleur correspondant.
- Ordonnancez vos instructions pour minimiser le nombre de cycles pris par le corps de boucle. Combien de cycles prend votre programme au total ?
- Humainement, comment diminuer encore plus plus le nombre de cycles ? [1] Donnez la version du code source correspondant au programme. À combien de cycles arrivez-vous ?

Exercice Bonus. Intuition du polyédrique

En supposant avoir du parallélisme illimité, trouvez un ordonnancement optimal pour les instructions du dernier nid de boucle du programme suivant :

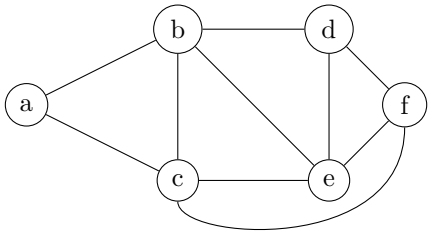
```
for (int i=0; i<N; i++)
      A[i,0] = I[i];
for (int j=1; j<N; j++)
      A[0,j] = I[j];

for (int i=1; i<N; i++)
      for (int j=1; j<N; j++)
S(i,j):    A[i,j] = A[i-1,j] + A[i,j-1];
```

Partie III - Allocation de registres

Exercice 4. Échauffement

En supposant n'avoir que 3 registres, appliquez l'algorithme de Chaitin sur le graphe d'interférence suivant :



Puis, en supposant que les nœuds c et d ont un arc d'affinité entre eux, ré-appliquez l'algorithme de Chaitin amélioré avec le critère de Georges.

Exercice 5. Optimalité du coloriage par algorithme glouton pour la forme SSA

Nous allons maintenant nous intéresser à l'allocation de registre sur une suite de basic-block n'incluant pas de boucle (au lieu de se limiter à un unique basic-block). On peut remarquer que la construction du graphe d'interférence et l'application de l'algorithme de Chaitin se font sans aucun problème.

1. Indice : il faut toucher à la boucle `for`

Préliminaire : Impact de la forme SSA sur le coloriage. Considérez le (bout de) programme suivant :

```
a = ...
b = ...
if (conditionquelconque) {
    c = ...
    d = ... b ...
    ... = c
} else {
    e = ...
    d = ... a ...
    ... = e
}
... = d
```

Questions :

- Étudiez les durées de vies des variables explicitées (`a`, ..., `e`) et construisez le graphe d'interférence. Combien de couleurs sont nécessaires pour colorer ce graphe (COLOR) ? Quel est le nombre de variables en vie simultanément (MAX_LIVE) ?
- Écrivez ce (bout de) programme sous forme SSA, puis répondez aux questions précédentes sur ce nouveau programme. Que remarquez-vous ?

Comme vous venez de le voir, passer un programme sous forme SSA peut relâcher les contraintes sur la coloration du graphe d'interférence et peut diminuer son degré chromatique. En fait, on va prouver que l'algorithme que coloration de graphe de Chaitin est optimal (en nombre de couleurs employés et en supposant qu'on spille les bons nœuds) si le code a été passé sous forme SSA au préalable.

Pour cela, il va falloir montrer deux choses :

- Le graphe d'interférence sur lequel on fait la coloration de graphe permet optimalité (ie, MAX_LIVE = COLOR)
- En sélectionnant des nœuds particulier à spiller, il est possible de colorer le graphe d'interférence précédent avec MAX_LIVE couleurs.

Partie 1 - Graphe d'interférence sous forme SSA. De base, on a les inégalités suivantes pour tout graphe : MAX_LIVE $\leq$ MAX_CLIQUE $\leq$ COLOR. Nous allons montrer en deux temps qu'il s'agit en fait d'égalités.

1) MAX_CLIQUE = COLOR : Un graphe *parfait* est, par définition, un graphe qui vérifie cette égalité. Montrons donc que le graphe d'interférence d'un programme sous forme SSA est forcément parfait.

Un graphe est *cordal* (aussi appelé *graphe triangulé*) si tous ses cycles de taille ≥ 4 admet une corde. Un *graphe d'intersection* est un graphe construit à partir d'une famille d'ensembles, tels que chaque ensemble correspond à un nœud du graphe et que le graphe admet une arête entre deux nœuds ssi leurs ensembles correspondants ont une intersection non vide.

Questions :

- (Théorème de Gavril) Montrez qu'un graphe d'intersection d'une famille de sous-arbre d'un arbre est un graphe cordal. [2]
- Faites le lien entre la forme SSA et les graphe d'intersection d'une famille de sous-arbre d'un arbre.
- [Exo bonus - admettez-le dans un premier temps] Montrez qu'un graphe cordal est parfait.
- Déduisez-en que MAX_CLIQUE = COLOR.

2) MAX_LIVE = MAX_CLIQUE : Là encore, il va falloir se servir de l'hypothèse que le graphe est un graphe d'interférence d'un programme sous forme SSA.

Questions :

- Montrez que si $u_1, \ldots, u_n$ forment une clique, alors $u_1, \ldots, u_n$ sont vivantes en même temps.
- Déduisez-en que MAX_LIVE = MAX_CLIQUE.

2. Ces deux ensembles sont même égaux, mais la preuve dans l'autre sens est beaucoup plus compliquée à démontrer

Partie 2 - Optimalité de la méthode gloutone. On appelle un *nœud simplicial* un nœud tel que ses voisins forment une clique. On suppose le lemme suivant : si G est cordal, alors G admet un nœud simplicial s et $G - \{s\}$ est encore cordal.

Questions :

- Montrez que l'algorithme de Chaitin (avec l'élimination des nœuds de degré $< k$) trouve bien une k coloration si $k \geq MAX\_LIVE$.

Références :

– Théorème de Gavril : *The Intersection Graphs of SUbtrees in Trees Are Exactly the chordal Graphs*, per F.Gavril, soumis en 1973.
– Thèse de Florent Bouchez : `http://florent.bouchez.free.fr/?page=recherche/these`