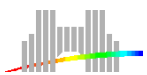
Master Informatique Fondamentale - M1 Compilation

Allocation de Registres

Paul Feautrier

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

13 avril 2007



Le point de départ

Une représentation intermédiaire en forme de code 3 adresses :

- $a \leftarrow b \otimes c$
- Les opérandes peuvent être :
  - des constantes
  - des *registres virtuels*
  - des cellules de mémoire, pour les données qui ne peuvent être mises en registre pour cause d'aliasing ou parce qu'elles sont visibles d'autres parties du programme.

Le problème, I

Allouer chaque registre virtuel à un registre physique.

- En respectant les contraintes de la machine cible : Classes de registres, Contraintes pair-impair, Recouvrements
- Il y a en général plus de registres virtuels que de registres physiques
- Quand un registre virtuel héberge plusieurs registres physiques, on doit écrire du *spill code* pour sauvegarder / restaurer la variable qu'il contient.
- La place nécessaire à la sauvegarde doit être prévue dans l'enregistrement d'activation
- minimiser la quantité de spill code

Le problème, II : questions de temps

On peut fractionner arbitrairement l'allocation de registres : il suffit de tout sauver en mémoire. L'allocation se fait par intervalles :

- par instruction. On n'alloue que les temporaires. Peu utilisé.
- par bloc de base. Seules doivent rester en mémoire les variables aliasées ainsi que les variables vivantes à l'entrée et à la sortie du bloc.
- par procédure. Seules doivent rester en mémoire les variables globales et les paramètres.
- pour tout le programme : peu utilisé car il n'y a pas en général assez de registres.

Allocation dans un bloc de base

Trois méthodes :

- Méthode top-down : on alloue de préférence les registres virtuels les plus fréquemment utilisés.
- Méthode de Sheldon Best : on simule le fonctionnement d'un cache parfait (algorithme de Belady)
- Méthode des intervalles de vivacité : on identifie les couples de registres virtuels qui ne peuvent pas partager le même registre physique

Méthode top-down

Principe :

- On compte les occurences de chaque registre virtuel, et on les classe par ordre de nombre d'occurences décroissantes.
- On réserve un certain nombre n de registres physiques (n dépend de l'architecture) pour les intermédiaires de calcul. n est de l'ordre de 1 à 3.
- S'il y a k registres physiques, on leur alloue les $k - n$ premiers registres virtuels
- Les autres registres virtuels sont alloué en mémoire. On corrige le code en conséquence.

Inconvénient : un registre virtuel peut être très utilisé au début d'un BB, puis inutile ensuite. Sa place n'est pas récupérée.

Algorithme de Sheldon Best

Principe : on simule le fonctionnement d'un cache parfait.

- Quand un registre virtuel est nécessaire, on le copie dans un registre physique.
- Si possible, on utilise un registre libre.
- Sinon, on recopie en mémoire le registre virtuel qui sera réutilisé le plus tard possible.
- La méthode a été prouvée optimale par Belady dans le contexte de la gestion de la mémoire virtuelle

Algorithme de Sheldon Best, structures de données

- $\ell(v, k)$ le numéro de la plus proche instruction utilisant le registre virtuel v après l'instruction k ou $\perp$
- $\phi(v)$ le numéro du registre physique contenant v ou $\perp$
- $\nu(r)$ le numéro du registre virtuel contenu dans r ou $\perp$
- La table ℓ est précalculée ; les deux autres tables sont initialisées à $\perp$

Programme principal

Data: N : le nombre d'instructions du BB
for $k = 1 \dots N$ **do**
    Let $v_1 \leftarrow f(v_2, v_3)$ *l'instruction* k;
    `fetch`(v_2, k) ;
    `fetch`(v_3, k) ;
    if $\ell(v_2, k) = \perp$ **then**
        `free`(v_2)
    end
    if $\ell(v_3, k) = \perp$ **then**
        `free`(v_3)
    end
    `fetch`(v_1, k) ;
    `emit(F` $\phi(v_1), \phi(v_2), \phi(v_3)$`)`;
end

Obtenir un registre

```
fetch(v,k);
if ∃r : ν(r) = v then
    return r
else
    r := allocate(k);
    ν(r) := v;
    φ(v) := r;
    emit(LOAD r, M[r]);
    return (r);
end
```

Allouer un registre

`allocate(`k`);`
if $\exists r : \nu(r) = \perp$ **then**
    return (r)
else
    Let $r : \ell(\nu(r), k) = \min_i \ell(\nu(i), k)$;
    `emit(STO` $r, M[r]$`);`
    return r
end

Libérer un registre

`free(`r`);`
;
$v = \nu(r)$;
$\nu(r) = \bot$;
$\phi(v) = \bot$;

L'algorithme peut être accéléré :

- ajouter une pile des registres libres
- ajouter une table hashcodée inverse de la table ν

Observations

- Bien que l'algorithme soit optimal, on peut parfois faire mieux en utilisant plus d'information : par exemple, en évitant de décharger les registres qui sont identiques à leur copie en mémoire (*propre*).
- Dans l'algorithme d'allocation, on peut soit choisir de préférence un registre propre, soit, parmi tous les registres ayant la valeur maximale de ℓ, donner la préférence à un registre propre.
- Les deux stratégies sont valables, mais aucune n'est meilleure que l'autre
- Il est possible de généraliser pour gérer plusieurs classes de registres

Allocation par coloriage de graphe

- Pour chaque valeur créé dans le BB, on défini un intervalle de vivacité, pendant lequel la valeur doit être conservée
- On alloue un registre physique à chaque valeur, en tenant compte du fait que deux intervalles disjoints peuvent partager le même registre
- Pour cela on construit un graphe d'interférence et on le colorie
- Chaque couleur correspond à un registre physique
- S'il y a trop de couleurs, on doit renvoyer une valeur en mémoire.

Intervalle de vivacité

- Soit k une instruction qui affecte une valeur au registre virtuel v.
- Soit k' l'écriture dans v qui suit immédiatement k, ou la fin du BB si v n'est plus modifié
- Soit ℓ la dernière lecture de v dans l'intervalle $[k, k']$. Si ℓ n'existe pas, c'est que l'écriture en k est inutile et peut être supprimée
- $[k, \ell]$ est un intervalle de vivacité

Renommage des registres virtuels

Lors de la sélection des instructions, il y a plusieurs stratégies pour engendrer les registres virtuels :

- On peut avoir attribué un nom unique à chaque RV. Dans ce cas, il y a correspondance entre les RV et les intervalles de vivacité
- Si on a réutilisé les RV, il peut y avoir plusieurs intervalles pour chaque RV. On peut renommer les RV pour qu'il y ait correspondance
- On peut aussi faire l'union des intervalles de vivacité de chaque RV. L'alloction est moins bonne, mais la complexité de l'algorithme diminue.

Graphe d'interférence

- Chaque sommet correspond à un intervalle de vivacité et à un registre virtuel
- Deux sommets sont adjacents si les intervalles correspondants ont une intersection non vide.
- Deux sommets adjacents ne peuvent pas utiliser le même registre physique.
- Attention, un registre est lu en début de cycle, écrit en fin de cycle. Donc les intervalles $[a, b]$ et $[b, c]$ n'interfèrent pas.

Coloriage d'un graphe

Colorier un graphe, c'est affecter à chaque sommet une couleur de telle façon que deux sommets adjacents aient des couleurs différentes.

- Si on peut colorier le graphe d'interférence avec k couleurs, alors il suffit de k registres pour implémenter le programme
- Le nombre minimum de couleurs nécessaires est le *nombre chromatique* du graphe.
- Calculer le nombre chromatique d'un graphe (ou déterminer si un graphe peut colorié avec k couleurs) est un problème NP-complet qui n'a pas de bonne approximation
- Il existe de nombreux algorithmes de coloriage, soit exacts (programmation 0/1, branch-and-bound) soit heuristiques.

Algorithme de coloriage, I

L'algorithme utilisé en compilation est basé sur l'observation suivante. Soit k le nombre de registres disponibles.

Si un sommet a moins de k voisins, il est toujours possible de le colorier quelles que soient les couleurs des voisins.

Première phase :

- Créer une pile vide
- Tant que tous les sommets n'ont pas été enlevés du graphe d'interférence
- Choisir un sommet ayant moins de k voisins (un sommet "libre")
- S'il n'en existe pas, choisir un sommet suivant un critère arbitraire (par exemple, le sommet correspondant à l'intervalle de vivacité ayant le moins de lectures)
- Enlever le sommet choisi ainsi que les arcs adjacents

Algorithme de coloriage, II

Deuxième phase.

- Initialiser un graphe d'interférence vide
- Tant que la pile n'est pas vide
- Extraire le sommet de pile et le réintégrer dans le graphe d'interférence avec les arcs adjacents
- Si ses voisins utilisent moins de k couleurs (et en particulier s'il a moins de k voisins), lui attribuer l'une des couleurs restantes
- Sinon, l'algorithme échoue.

Spill code

L'algorithme échoue s'il n'a pas assez de registres pour colorier le graphe. Il y a plusieurs stratégies possibles :

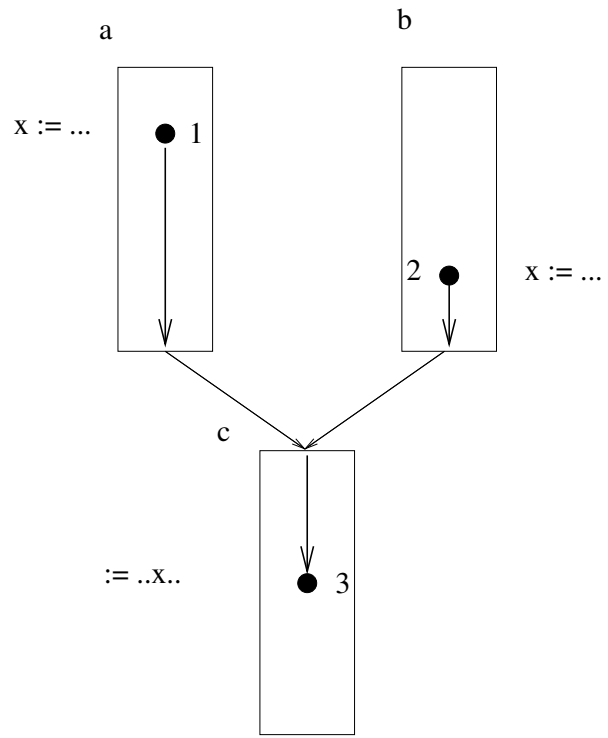
- Si l'on a réservé suffisamment de registres, on peut réécrire l'intervalle en cause comme si la valeur considérée était en mémoire. On "neutralise" l'intervalle de vivacité et on poursuit.
- On peut "neutraliser" l'intervalle de vivacité, puis recalculer le graphe d'interférence et recommencer.
- On peut essayer de couper en deux l'intervalle de vivacité à l'aide d'une seule écriture / lecture, et recommencer.

Au delà d'un bloc de base

Le problème : étendre l'algorithme d'allocation en présence de structures de contrôles complexes

- Les régions de vivacité ne sont plus des intervalles, mais des unions d'intervalles, et leur calcul est plus complexe
- Pour choisir la position du *spill code*, il faut tenir compte des fréquences d'exécution
- Hypothèse : on travaille à partir du graphe des blocs de base de la procédure.

Structure d'un intervalle de vivacité



- Il ya deux intervalles de vivacité : $[1, 3]$ et $[2, 3]$
- Comme ces deux intervalles ont un segment commun, il faut utiliser le même registre physique ou prévoir des copies
- On forme donc un seul domaine de vivacité
- On observe que x est "vivante" en sortie de A, de B, et vivante en entrée de C.

Quelques définitions

Une variable est vivante en un point du programme s'il existe un chemin vers une lecture qui ne passe par aucune écriture.

- Une variable x est *exposée* dans un bloc de base b ($x \in \mathrm{Exp}(b)$) ssi sa première lecture précède sa première écriture
- Un bloc de base b est *transparent* pour une variable x ($x \in \mathrm{Trans}(b)$) ssi b ne contient aucune écriture de x.

Equations de vivacité

- Une variable est vivante en début d'un bloc ssi elle est exposée dans le bloc ou si le bloc est transparent pour elle et elle est vivante en fin de bloc :

$$\mathrm{LiveIn}(b) = \mathrm{Exp}(b) \cup (\mathrm{Trans}(b) \cap \mathrm{LiveOut}(b))$$

- Une variable est vivante en fin d'un bloc si elle est vivante au début de l'un de ses successeurs :

$$\mathrm{LiveOut}(b) = \cup_{s \in \mathrm{succ}(b)} \mathrm{LiveIn}(\mathrm{s})$$

- On peut d'ailleurs éliminer LiveIn et ne conserver que $\mathrm{LiveOut}$

Résolution itérative

Les équations de vivacité sont de forme itérative. On les résoud par recherche de point fixe :

- Initialiser tous les $\mathrm{LiveOut}$ à l'ensemble des variables du programme
- Initialiser une FIFO avec tous les blocs de base du programme
- Tant que la FIFO n'est pas vide :
  - extraire un bloc de base, b
  - $L = \mathrm{LiveOut}(b)$
  - recalculer $\mathrm{LiveOut}(b)$ à l'aide de la formule ci-dessus
  - Si $\mathrm{LiveOut}(b) \neq L$, ajouter à la FIFO tous les prédécesseurs de b

La preuve de convergence et de correction sera vue plus tard.

Construction des domaines de vivacité

- On construit les intervalles internes aux BB comme précédemment
- Pour toute variable LiveIn on créé un intervalle allant de l'entrée du bloc à la dernière lecture avant écriture ou à la fin du bloc
- Pour toute variable $\mathrm{LiveOut}$ on crée un intervalle allant de la dernière écriture ou du début du bloc jusqu'à la fin du bloc
- Deux intervalles font partie d'un même domaine si l'un se termine à la fin du bloc a et l'autre commence au début de b et si b est un successeur de a
- Les domaines sont les classes d'équivalence de la fermeture transitive de la relation ci-dessus. On renomme les registres virtuels par domaine
- On peut construire ensuite un graphe d'interférence.

Sélection du spill code

Dans l'algorithme de coloriage, on doit tenir compte de la fréquence d'exécution de chaque domaine.

- On peut profiler le programme
- On peut procéder à une analyse statique, en général approchée (exemple : chaque boucle tourne 10 fois)
- On associe à chaque domaine un poids (exemple : le produit de sa fréquence d'exécution par le nombre de références)
- Quand il n'y a pas de domaine "libre", on sélectionne celui de plus faible poids restant.

Allocation interprocédurale

L'allocation interprocédurale est complexe et peu utile.

- Le processeur doit posséder un grand nombre de registres (typiquement, la majorité des procédures doivent avoir été coloriées sans *spill*
- L'allocation interprocédurale ne concerne que les variables globales du programme
- On peut par exemple, après avoir déterminé le nombre de registres inutilisés, procéder comme ci-dessus en remplaçant le graphe de contrôle par le graphe des appels de procédures.