Compilation

TP 7 : Génération de code

G. Iooss & C. Alias

Dans ce TP, nous abordons la dernière partie de la compilation: la production finale du code Digmips. Comme on l'a vu en cours, elle se décompose en trois étapes: (i) sélection des instructions Digmips à partir du DAG de chaque bloc de base, (ii) ordonnancement des instructions, et (iii) allocation des temporaires.

Exercice 1. *Sélection, ordonnancement*

La première étape est de choisir les instructions Digmips pour évaluer le DAG de chaque bloc de base, et de trouver un ordre d'exécution (ordonnancement) convenable.
Récuperez le compilateur complet `digcc.tgz` et décompactez le.
Ouvrez le fichier Backend.cc.

- **backend()** (ligne 228) part de la représentation intermédiaire passée en paramètre (cfg) et produit le code final en appliquant successivement les trois étapes décrites ci-dessus.
- **select()** (ligne 199) réalise les deux premières étapes. Pour chaque bloc de base, elle produit un DAG (ligne 210) et sélectionne/ordonnance les instructions Digmips correspondantes avec la fonction select_dag (ligne 216). Le résultat est un CFG dont les noeuds sont des instructions Digmips, pour lequel il faut calculer les durées de vie des temporaires (ligne 221).
- **select_dag()** (ligne 183) visite chaque racine du DAG, et produit le code avec un parcours en profondeur à gauche d'abord par un appel à select_root (ligne 195). L'ordonnancement actuel est donc direct, sans tentative d'optimisation.
- **select_root()** (ligne 20) effectue la sélection à proprement parler. Comme le grain des instructions Digmips est plus petit que celui de la représentation intermédiaire (pourquoi?), la technique du tuilage ne s'applique pas, et il faut simplement traduire chaque noeud du DAG. Cette traduction est effectuée par la fonction digmips_translate (ligne 162), qui ajoute les nouvelles instructions Digmips au CFG résultant, digmips_cfg. Naturellement, il ne faut pas oublier de copier le résultat d'un noeud dans chacun de ses registres marqués comme liveout (car utilisés en dehors du bloc courant).

Ouvrez le fichier CodeDigmips.h.
La classe CodeDigmips décrit les instructions Digmips. Elle hérite de Code, ce qui permet de construire un CFG d'instructions Digmips et de calculer les durées de vie des temporaires en utilisant la méthode de CFG. Le fichier comporte également:

- des constructeurs (lignes 82–100) qui ajoutent une instruction au cfg passé en paramètre,
- des macros (lignes 103–114) qui produisent la séquence d'instruction Digmips équivalente pour plusieurs pseudo-instructions,
- les fonctions principales de traduction (lignes 121–fin).

Manip.

- Dans le fichier Backend.cc:
  1. décommentez les lignes 230–245 (pour afficher le DAG de chaque bloc de base)
  2. décommentez la ligne 254 (pour afficher le code Digmips produit la sélection/ordonnancement
  3. commentez la ligne 309 (pour ne pas afficher le code final).

  Compilez et testez sur tests/simple.c.
- On s'intéresse au bloc qui commence par alloc_0 (deuxième DAG, lignes 19–36). Etudiez la traduction. Comment se traduit r14 = 1? Comment l'expliquez vous?
- Rémettez le fichier dans son état initial.

Exercice 2. *Allocation*

Le code produit par l'étape précédente manipule encore des temporaires. Il faut maintenant leur allouer des registres et des emplacements dans la pile. Cette étape est réalisée dans Backend.cc, lignes 260–281.

Manip.

- La dernière étape de la fonction select est de calculer les durées de vie des temporaires. Affichez donc le résultat en décommentant la ligne 255. Testez à nouveau sur tests/simple.c. Re-commentez la ligne 255.
- A partir des ces informations, on construit un graphe d'interférence (ligne 263). Il suffit de déclarer en conflit deux temporaires présents en même temps soit dans le live-in d'une instruction (InterferenceGraph.cc, lignes 19–42) soit dans son live-out (InterferenceGraph.cc, lignes 45–68). Afficher le graphe d'interférence en décommentant la ligne 265. Notez que certains temporaires réservés (r0, r1, r6, r7) n'y figurent pas. Normal, ils sont déjà alloués aux registres physiques correspondants. Re-commentez la ligne 265.

Pour allouer des registres (et des emplacements pile aussi), il faut maintenant colorier ce graphe. Comme on tient à ce que le temporaire r2 soit logé dans le registre r2, on précolorie le graphe (ligne 268)[1]. On lance ensuite l'allocation sur 4 registres (ligne 270). En effet, r0 et r1 sont réservés pour le spill, et r6,r7 pour la pile. Reste donc r2, r3, r4, r5 pour les affaires courantes (que l'allocateur nomme 0,1,2,3... d'où le shift lignes 273–276).

Ouvrez le fichier Allocation.h. La classe Allocation contient le mapping final temporaire → temporaire "physique" (registre ou pile).

Ouvrez le fichier Allocation.cc. Le constructeur commence ligne 146. On utilise l'algorithme de Chaitin pour construire notre allocation. Il faut donc un mécanisme pour retirer/ajouter des noeuds au graphe d'interférence... On utilise simplement une liste de noeuds existants, nodes, qu'on initialise avec tous les noeuds (lignes 148–151) et que l'allocateur (ligne 162) mettra à jour. L'allocateur (chaitin, lignes 74–) prend en paramètre le graphe d'interférence, la liste des noeuds existants et le nombre K de registres physiques. Il s'agit d'une implémentation directe de l'algorithme vu en cours. Remarquez l'appel récursif sur $G - \{s\}$ lignes 97–99. Une fois les registres alloués, il faut allouer les emplacements de pile pour les temporaires spillés (lignes 164–). Il suffit de colorier avec $+\infty$ couleurs la restriction du graphe d'interférence aux noeuds spillés. Pour calculer cette restriction, il suffit de ne faire exister que les noeuds spillés (lignes 168–176) et de ré-appeler l'allocateur (ligne 186). Le numéro assigné par Chaitin peut alors être vu comme le décalage du temporaire dans la pile...

Manip.

- Dans Backend.cc, décommentez la ligne 279 pour afficher l'allocation. Testez sur tests/simple.c.

Exercice 3. *Emission de code*

Bon, maintenant, on a vraiment tout ce qui nous faut: le code digmips avec temporaires et l'allocation des temporaires. Il ne reste qu'à appliquer l'allocation... sans oublier de produire du *spill-code* chaque fois qu'un temporaire est alloué dans la pile. Cette étape est réalisée par les lignes 282–303 dans Backend.cc. Pour chaque instruction produite à l'étape 1, la fonction digmips_apply (ligne 301) produit la version "allouée".

Ouvrez le fichier CodeDigmips.cc, allez à la ligne 312. On commence par remplacer les temporaires chanceux par les vrais registres correspondants (lignes 333–349). Ensuite on produit le code en insérant le *spill-code* des registres lus (lignes 355–382), on insère l'instruction "patchée" avec de vrais registres (ligne 384) et on insère le *spill-code* du registre écrit quand c'est nécessaire (lignes 387–388).

Manip.

- Produire le code final sur tests/simple.c.
- En vous aidant du code produit dans l'exercice 1 et de l'allocation, constatez son application et la production de *spill-code*.
- Comment/pourquoi évite-t-on le spill-code dans le prélude et dans le postlude d'une fonction?

[1] avec r0, mais qui deviendra r2 en fait, cf. lignes 273–276

Exercice 4. *Have fun!*

Amusez à compiler/exécuter de petits programmes. Exemple: somme des n premiers entiers, rectangle d'étoiles, ...