Projet de Compilation Avancé

Mathieu Chailloux Matthieu Dien

2 mai 2013

Le but de ce projet est la prise en main des techniques d'optimisation de code assembleur (ici MIPS) et la mise en pratique des concepts appris en cours. Pour cela, il nous a fallu manipuler les notions de dépendances, entre blocs ou entre instructions, de graphe de flot de contrôle, de graphe de flot de données, ou encore de chemin critique. Cela nous a permis de pouvoir atteindre l'étape de réordonnancement du code qui est la première véritable technique d'optimisation. Par la suite, nous allons détailler les fonctionnalités implémentées.

1 Reconnaissance de fonctions

Cela consiste en parcourir le programme et identifier les fonctions. Celles-ci sont délimitées par les directive ".ent" et ".end". Nous avons donc simplement créé une nouvelle fonction (de la classe Function) à chaque directive ".ent" rencontrée, et situer la fin de celle-ci à la prochaine directive ".end". Voir la fonction correspondante "comput_function" dans la classe Program.

2 Reconnaissance de blocs de base

Pour chaque fonction, on veut identifier les blocs de base. On parcourt donc les lignes de la fonction en identifiant les débuts et fins de bloc selon les règles suivantes :

- Un label est un début de bloc, il signifie donc que la ligne précédente est une fin de bloc.
- Une instruction de type branchement (donc un saut) signifie la fin d'un bloc mais elle est suivi d'un delayed slot, la fin du bloc est donc l'instruction suivante.

Cela correspond à la fonction "comput_basic_block" de la classe Function.

3 Dépendances entre blocs de base

On s'intéresse maintenant aux dépendances entre blocs de base. Les blocs successeurs d'un bloc sont déterminés ainsi :

- Si le bloc d'origine ne se termine pas par un branchement, alors l'unique successeur est le bloc suivant dans le code

– Si le bloc se termine par un branchement conditionnel, alors la cible du saut (le bloc qui commence par le label visé dans le saut) et le bloc suivant dans le code sont ses successeurs
– Si le bloc se termine par un branchement inconditionnel, alors son unique successeur est la cible du saut
– Enfin, cela ne s'applique qu'aux sauts directs, les sauts indirects (dont la cible est contenue dans un registre par exemple) ne sont pas concernés par ces règles et leur comportement n'est analysable dans ce cadre.

Voir "comput_succ_pred_BB" dans la classe Function.

4 Construction des graphes de flot de contrôle

La construction de ces graphes étant déjà écrites dans le code donné, le seul travail à faire ici a été de parcourir les fonctions d'un programme et, pour chacune, d'ajouter un nouveau CFG prenant en paramètre le premier bloc de base de la fonction, ainsi que son nombre de bloc de base.

Voir "comput_cfg" dans le classe Program.

5 Dépendances entre instructions

On s'intéresse ici aux dépendances entre instructions appartenant à un même bloc de base. Pour cela, on vérifie pour chaque couple d'instruction s'il existe une dépendance entre elles (la fonction faisant cette vérification étant déjà implémentée) et le cas échéant on ajoute le successeur et le prédécesseur adéquat. Enfin, il faut refaire une itération pour ajouter une dépendance de contrôle entre chaque instruction qui n'a pas de successeur (dont personne ne dépend) avec le saut de fin de bloc, s'il y en a un. Ce traitement est fait dans la fonction "comput_pred_succ_dep" de la classe Basic_block.

6 Construction du graphe de flot de données

Une fois les dépendances entre instructions calculées, la construction du graphe de flot de données (DFG) d'un bloc est assez naturel. On parcourt une première fois les instructions pour créer les noeuds (de la classe Node_dfg) les représentant et pour repérer le noeud correspondant au saut de fin de bloc s'il y en a un. On fait ensuite une deuxième itération pour effectuer le traitement suivant sur les noeuds :

– Si le noeud n'a pas de prédécesseur (son instruction ne dépend d'aucune autre), alors on le rajouter à la liste des racines, "_roots".
– Si le noeud représente un saut, alors c'est le saut de fin de bloc et on ajoute le noeud suivant à la liste "_delayed_slot"
– Pour chaque successeur de l'instruction, on va chercher le noeud représentant ce successeur, et on ajoute un nouvel arc avec la dépendance concernée et le délai qu'elle occasionne.

Voir le constructeur de la classe Dfg.

7 Calcul du chemin critique

Le calcu du chemin critique se fait selon l'algorithme vu en cours. Une méthode `get_inverser_topoligical_order` a été ajouté pour calculer l'ordre topologique nécessaire à la construction du chemin critique. L'algorithme est simple : on part d'une liste temporaire des noeuds sans successeurs. Pour chaque noeud dans la liste temporaire (en partant du premier), on l'ajoute en queue à la liste finale, puis on ajoute ses prédécesseurs en queue de la liste temporaire. On itère ce processus jusqu'à ce que la liste temporaire soit vide.

8 Réordonnancement

Le réordonnancement s'effectue selon l'algorithme de liste vu en cours, nous ne nous attarderons donc pas sur l'algorithme en lui-même. Pour le traitement, on s'appuie sur la liste "_inst_ready" qui contient les instructions prêtes à être traiter. Au début, cette liste doit contenir les racines. Tant qu'il reste des élements à cette liste, il faut continuer le traitement qui consiste en déterminer laquelle de ces instructions on veut choisir. Pour cela, on va trier la liste (en s'appuyant sur la fonction "sort" des listes) en ordre croissant de priorité, c'est à dire que l'on trie d'abord selon les index, pour finir par trier par les poids. La seule priorité un peu différente est celle qui vérifie qu'on ne cause pas de cycle de gel. Pour celle-là, il faut vérifier que pour chaque instruction réordonnancée, le délai avec l'instruction en cours de traitement n'excède pas la différence d'index entre celles-ci.

Voir la fonction "shceduling" de la classe Dfg.