Projet CA (MI190) Rapport

Toutes les fonctions des TME ont été réalisées. Il s'agit du découpage du programme en fonctions, puis des fonctions en blocs de base. Ensuite, les liens, prédécesseurs et successeurs, entre les blocs de base, c'est-à-dire le calcul du graphe de flot de contrôle (CFG).

Nous détaillons ici les fonctions demandées pour la suite du projet.

Contenu

Question 1 : construction du graphe de flot de données (DFG) associé à un bloc de base

Pour cette question, nous avons écrit la fonction compute_pred_succ_dep de la classe Basic_block. Celle-ci parcourt toutes les instructions de la dernière à la première en recherchant les dépendances RAW, WAR et WAW puis les dépendances mémoire et de contrôle.

Puis dans la classe Dfg nous avons remplis le constructeur. Tout d'abord nous vérifions la présence de delayed slot, si il y en a un, il faudra stopper notre recherche un cran plus tôt au niveau du saut.
Ensuite pour chaque instruction nous vérifions ces dépendances avec ces prédécesseurs.
Si une instruction n'a pas de prédécesseur et que ce n'est pas un delayed slot alors on l'ajoute à la liste des racines.
Par la suite, pour chaque successeur, on trouve le délai correspondant au type de la dépendance. Une fois le délai connu, on ajoute un arc de dépendance entre les 2 nœuds.

Question 2 : poids du chemin critique

La fonction comput_critical_path et ses auxiliaires permettent de calculer le poids du chemin critique.
On initialise le poids des nœuds du delayed slot, s'ils existent à 2, sinon à 1, par la suite on lance une fonction récursive à partir des racines.
Celle-ci calcule le poids de chaque nœud comme le maximum des poids de ses successeurs additionnés aux délais des arcs vers ces successeurs.

Il est ensuite facile de réaliser get_critical_path qui renvoie le poids maximum parmi les racines.

Question 3 : ordonnancement des instructions d'un bloc de base

Scheduling

Pour réaliser l'ordonnancement des instructions, il fallait tout d'abord connaître le nombre de descendants (pas seulement les successeurs directs) de chaque nœud. Nous avons donc coder la fonction compute_nb_descendant dans la classe Dfg qui fait ce travail.

L'algorithme implémenté est celui donné en cours. Dans la fonction scheduling, on maintient la liste _inst_ready qui contient les instructions prêtes.
A chaque tour de boucle, on choisit celle qui est lancée, en suivant les règles. A chaque règle, on insère dans une liste temporaire les instructions qui satisfont la règle. S'il n'y en qu'une, alors c'est celle-là.
*Sinon, on passe à la règle suivante. Pour les étapes nécessitant de trier les instructions selon un critère, des fonctions compare*** sont présentes juste au-dessus.*

De plus, nous avons codé une fonction add_node_now_ready qui ajoute les nœuds dans la liste lorsqu'ils deviennent prêts. On utilise pour cela un champ _traitee que nous avons ajouté dans la classe Node_dfg et qui indique si un nœud a été traité ou non. Si le nœud est traité, la valeur du champ est le numéro de cycle auquel l'instruction a été lancée.

Nombre de cycles

La fonction nb_cycles de la classe Dfg a été réalisée. Elle se sert également du champ _traitee (tous remis à -1). On parcourt toutes les instructions de new_order en calculant les délais induits les cycles de gel.

Exemples d'exécutions

Ré ordonnancement (./bin/cpp/test_karine_dfg ./src/examples/test_asm32.s)

Ordre classique :

- *i0 addiu $29,$29,65520*
- *i1 sw $30,12($29)*
- *i2 or $30,$29,$0*
- *i3 sw $4,16($30)*
- *i4 sw $5,20($30)*
- *i5 sw $6,24($30)*
- *i6 sw $7,28($30)*
- *i7 sw $0,0($30)*
- *i8 j $l2*
- *i9 add $0,$0,$0*

Scheduled Order :

- *i0: addiu $29,$29,65520*
- *i1: sw $30,12($29)*
- *i2: or $30,$29,$0*
- *i7: sw $0,0($30)*
- *i6: sw $7,28($30)*
- *i5: sw $6,24($30)*
- *i4: sw $5,20($30)*
- *i3: sw $4,16($30)*
- *i8: j $l2*
- *i9: add $0,$0,$0*

Trace de l'exécution :

➔ Instructions ready :
- i0
- Fin

➢ Règle 2 -> Instruction choisie : i0

➔ Instructions ready :
- i1
- i2
- i3
- i4
- i5
- i6
- i7
- i8
- Fin

➢ Règle 2 -> Instruction choisie : i1

➔ Instructions ready :
- i2
- i3
- i4
- i5
- i6
- i7
- i8
- Fin

➢ Règle 2 -> Instruction choisie : i2

➔ Instructions ready :
- i3
- i4
- i5
- i6
- i7
- i8
- Fin

➢ Règle 6 -> Instruction choisie : i7

➔ Instructions ready :
- i3
- i4
- i5
- i6
- i8
- Fin

➢ Règle 6 -> Instruction choisie : i6

➔ Instructions ready :
- i3
- i4
- i5
- i8
- Fin

➢ Règle 6 -> Instruction choisie : i5

➔ Instructions ready :
- i3
- i4
- i8
- Fin

➢ Règle 6 -> Instruction choisie : i4

➔ Instructions ready :
- i3
- i8
- Fin

➢ Règle 4 -> Instruction choisie : i3

➔ Instructions ready :
- i8
- Fin

➢ Règle 2 -> Instruction choisie : i8

➔ Instructions ready :
- i9
- Fin

➢ Règle 2 -> Instruction choisie : i9

Question 4 : renommage de registre

Renommage de registre (./bin/cpp/test_karine ./src/examples/test_asm32.s)

Mode opératoire

Le renommage se divise en 2 fonctions.

Tout d'abord une fonction auxiliaire (renomme) qui prends en argument le registre à modifier, l'instruction ou il commence, celle où il finit ainsi qu'un registre libre.
La fonction renomme dans la 1er instruction (la dernière en ordre d'exécution) seulement les paramètre, dans celle de fin (la première en ordre d'exécution) seulement le registre de sauvegarde.
Pour toutes les autres à chaque fois que le registre à modifier apparait, il le remplace par celui qui était libre.

La 2ème fonction (register_rename) parcourt la liste des instructions de la dernière à la première, si l'instruction écrit dans un registre alors pour toutes les instructions qui la précèdent, on vérifie si précédemment on n'écrit pas dans le même registre. Si oui l'on renomme (par le biais de la fonction auxiliaire) de cette instruction à la 1ère instruction du bloc de base. De ce fait, à chaque itération l'on modifie les registres précèdent lorsqu'il y a une « collision ».

La fonction renomme possédant certaines particularité en fonction de l'emplacement de l'instruction (1er ou dernière), cela nous permet de garder une cohérence globale et de ne pas utiliser en paramètre le même registre que celui qu'on définit.

Renommage BB – 0

- Avant
  - Begin BB
  - $l3:
  - i0 lw $2,0($30)
  - i1 sll $3,$2,2
  - i2 lw $2,24($30)
  - i3 addu $5,$2,$3
  - i4 lw $2,0($30)
  - i5 sll $3,$2,2
  - i6 lw $2,16($30)
  - i7 addu $2,$2,$3
  - i8 lw $4,0($2)
  - i9 lw $2,0($30)
  - i10 sll $3,$2,2
  - i11 lw $2,20($30)
  - i12 addu $2,$2,$3
  - i13 lw $2,0($2)
  - i14 addu $2,$4,$2
  - i15 sw $2,0($5)
  - i16 lw $2,0($30)
  - i17 addiu $2,$2,1
  - i18 sw $2,0($30)
  - End BB

- Apres
  - Begin BB
  - $l3:
  - i0 lw $41,0($30)
  - i1 sll $42,$41,2
  - i2 lw $41,24($30)
  - i3 addu $5,$41,$42
  - i4 lw $41,0($30)
  - i5 sll $38,$41,2
  - i6 lw $40,16($30)
  - i7 addu $39,$40,$38
  - i8 lw $4,0($39)
  - i9 lw $37,0($30)
  - i10 sll $3,$37,2
  - i11 lw $36,20($30)
  - i12 addu $35,$36,$3
  - i13 lw $34,0($35)
  - i14 addu $33,$4,$34
  - i15 sw $33,0($5)
  - i16 lw $32,0($30)
  - i17 addiu $2,$32,1
  - i18 sw $2,0($30)
  - End BB