Vincent Havinh

William Sergeant

Objectif du projet

Notre objectif pour ce projet est d'implémenter un ensemble de fonctions permettant de traiter du code assembleur. Ainsi, nous structurerons le code en délimitant les blocs de bases puis nous analyserons les dépendances entre les instructions. Enfin, nous apporterons des améliorations au code source telle que l'optimisation de l'ordonnancement des instructions et le renommage des registres afin de limiter les dépendances.

Fonctions implémentées

comput_basic_block()

Afin de délimiter les blocs de bases, nous parcourons chaque ligne du code correspondant à une instruction et analysons son type :

Si c'est une directive alors elle est ignorée et nous passons à l'instruction suivante

Si c'est un label il s'agit du début d'un nouveau bloc de base. Ainsi on ferme le bloc de base courant en lui donnant pour fin la ligne précédente et un nouveau est ouvert.

Si c'est une instruction et qu'elle est du type branchement alors c'est la fin du bloc courant, le delayed slot suivant le saut est ajouté et le bloc de base est fermé.

comput_succ_pred_BB()

Cette fonction sert à définir les liens de succession et précédence entre les blocs de base. Nous parcourons ainsi tous les blocs définis grâce à la fonction comput_basic_block() et calculons ses successeurs potentiels.

Afin de savoir si celui-ci en possède un, deux ou aucun nous analysons sa dernière instruction :

- Si c'est un saut inconditionnel, un appel de fonction ou qu'il n'y a simplement pas de saut, il aura un successeur qui sera le bloc suivant.
- Si c'est un saut conditionnel alors il aura deux successeurs : la cible du saut si la condition est respectée, le bloc suivant sinon.
- Si c'est un saut indirect ou qu'il s'agit du dernier bloc de base du programme (la dernière instruction étant JR R31), alors ils n'ont pas de successeur.

comput_CFG()

La fonction comput_CFG() étant déjà implémentée dans le code du programme de test, nous n'avons pas ré-implémenté sur cette implémentation.

compute_dom()

Nous utilisons l'algorithme vu en cours pour calculer les relations de dominance entre les blocs de bases. Tant qu'un changement de dominance a lieu dans les prédécesseurs d'un bloc, on recommence l'algorithme sur ses successeurs.

A noter que la racine se domine elle-même.

comput_pred_succ_dep()

Nous cherchons ici à définir les dépendances entre deux instructions au sein d'un même bloc de base.

Des prédicats servant à tester une dépendance donnée entre deux instructions nous permettent de déterminer aisément s'il en existe et le cas échéant, de trouver son type. Nous pouvons ainsi dresser la table des dépendances RAW1-2, WAR1-2 et MEM au sein du bloc.

En outre il faut ajouter les dépendances de contrôles avec les branchements s'ils sont présents.

La subtilité en ce point est de faire un bon usage des dépendances WAW, elles sont utilisées afin de stopper la recherche de dépendance avec l'instruction courante. En effet dans le cas d'une telle réécriture c'est la dernière instruction qui sera considérée pour les futures recherches de dépendances par rapport au registre destination.

nb_cycles()

Le but est à présent de calculer le nombre de cycles nécessaires à l'exécution d'un bloc de base. On s'intéresse alors au nombre de cycle de chaque instruction constituant le bloc.

A cet effet nous utiliserons la formule suivante :

Soit k l'instruction courante, soit p l'ensemble des instructions dont dépend k

$$cycle(k) = max(cycle(k-1)+1, max(cycle(p)+delai(pi,k))$$

Le tableau de délai entre les instructions nous a été fourni en annexe.

L'enjeu est de considérer les phases d'attente des instructions soumises dépendantes d'instructions précédentes (les cycles de gel).

compute_use_def()

Afin de connaître les registres vivants en différents points du programme, nous calculons les domaines DEF et USE de chaque bloc de base. Lorsqu'un registre est utilisé en écriture, il appartient au domaine DEF du bloc. Aussi, si un registre est accédé en lecture lors d'une instruction sans avoir été défini par une instruction antérieure dans le bloc, il appartiendra au domaine USE.

En outre certaines spécificités apparaissent lors de l'appel de fonction :

Le registre $2 contiendra le résultat de retour de la fonction, et le registre $31 contiendra lui l'adresse de retour de la fonction. Les registres $4,$5,$6,$7 contiennent les paramètres de l'appel de la fonction . Ces registres ne sont pas explicitement utilisés dans les instructions mais doivent être pris en compte afin de correctement définir les domaines DEF et USE :

$2 et $31 appartient à DEF du bloc.

$4, $5, $6,$7, s'ils ne sont pas définis avant, appartiennent à USE du bloc.

compute_live_var()

Cette fonction utilise les domaines DEF et USE fournis par la fonction précédente afin d'analyser la vivacité des registres en entrée et sortie de chaque bloc.

Un bloc aura comme liste de registres vivants Live-Out tous les registres présents dans la liste Live-In de ses successeurs. Si un bloc ne possède pas de successeur mais qu'il s'agit du bloc de fin du main ou du bloc de sortie d'une fonction, les registres $2 (contenant la valeur de retour de la fonction) et le registre $29 (pointeur de pile) seront vivants en sortie de bloc et doivent être pris en compte.

La liste Live-In d'un bloc est construite en y ajoutant :

- Les registres présents dans son Live-Out à l'exception de ceux définis dans le bloc
- Les registres appartenant au domaine USE du bloc.

On itère cette méthode sur tous les blocs de base, dans l'ordre inverse de leur ordre naturel de définition dans le programme source.

Le cas particulier des arcs retour nous contraint à effectuer plusieurs passes afin de prendre en compte le Live-In d'un bloc analysé ultérieurement dans la définition du Live-Out de son bloc prédécesseur (car l'analyse commence par les derniers blocs et remonte le flow des instructions à l'envers). Enfin, nous propageons cette nouvelle définition du Live-Out à son Live-In puis aux Live-out concernés, etc...

compute_def_liveout()

Dans le cas où un registre est vivant en sortie de bloc et que ce registre appartient au domaine DEF du bloc, nous cherchons à savoir quelle est la dernière instruction à l'avoir défini.

Ainsi, après avoir repéré les registres remplissant ces deux conditions, nous parcourons les instructions et on note celle qui définit le registre concerné. Si l'on rencontre une instruction le définissant à nouveau, l'information est mise à jour. Alors après une passe c'est bien la dernière instruction ayant défini le registre qui est retenue.

reg_rename()

Cette fonction de renomme les registres définis dans le bloc mais non vivants en sortie.

Nous cherchons d'abord à définir quels registres peuvent être renommés.

Pour cela une liste est construite contenant les registres absents du Live-Out du bloc mais définis dans le bloc.

Ensuite, nous itérons sur cette liste en repérant la première définition du registre à renommer puis nous effectuons le changement de nom dans celle-ci. Enfin nous parcourons les instructions suivantes en renommant les occurrences ultérieures des utilisations du registre jusqu'à atteindre la fin du bloc ou une nouvelle définition de ce registre.

Analyses expérimentales

Analyse de dep_inst3.s (BBO analysé en TD)

Soit le bloc de base suivant et le nombre de cycle nécessaires à son exécution (on utilise la table de délais initiale):

main:

```
i0 lw $4,0($6)
i1 lw $2,0($4)
i2 add $5,$14,$2
i3 ori $10,$6,0
i4 sw $5,0($10)
i5 lw $2,65524($10)
i6 addi $5,$2,4
i7 bne $5,$2,$15
i8 add $0,$0,$0
```

Instruction	Code	Cycle de sortie
0	lw $4,0($6)	1
1	lw $2,0($4)	3
2	add $5,$14,$2	5
3	ori $10,$6,0	6
4	sw $5,0($10)	7
5	lw $2,65524($10)	8
6	addi $5,$2,4	10
7	bne $5,$2,$l5	12
8	add $0,$0,$0	13

Total de cycles : 13

RENOMMAGE

Instruction	Code	Cycle de sortie
0	lw $9,0($6)	1
1	lw $2,0($9)	3
2	add $11,$14,$2	5
3	ori $10,$6,0	6
4	sw $11,0($10)	7
5	lw $2,65524($10)	8
6	addi $5,$2,4	10
7	bne $5,$2,$l5	12
8	add $0,$0,$0	13

Total de cycles : 13

ORDONNANCEMENT (sans renommage)

Instruction	Code	Cycle de sortie
0	lw $4,0($6)	1
3	ori $10,$6,0	5
1	lw $2,0($4)	2
2	add $5,$14,$2	3
5	lw $2,65524($10)	7
4	sw $5,0($10)	6
6	addi $5,$2,4	8
7	bne $5,$2,$l5	10
8	add $0,$0,$0	11

Total de cycles : 11

RENOMMAGE & ORDONNANCEMENT

Instruction	Code	Cycle de sortie
0	lw $9,0($6)	1
3	ori $10,$6,0	5
1	lw $2,0($9)	2
2	add $11,$14,$2	3
5	lw $2,65524($10)	7
4	sw $11,0($10)	6
6	addi $5,$2,4	8
7	bne $5,$2,$l5	10
8	add $0,$0,$0	11

On observe bien un gain de cycle avec l'opération de renommage & réordonnancement.

Soit le bloc de base suivant et le nombre de cycle nécessaire à son exécution (on utilise la table deuxième table de délais):

main:

```
i0 lw $4,0($6)
i1 lw $2,0($4)
i2 add $5,$14,$2
i3 ori $10,$6,0
i4 sw $5,0($10)
i5 lw $2,65524($10)
i6 addi $5,$2,4
i7 bne $5,$2,$15
i8 add $0,$0,$0
```

Instruction	Code	Cycle de sortie
0	lw $4,0($6)	1
1	lw $2,0($4)	4
2	add $5,$14,$2	7
3	ori $10,$6,0	8
4	sw $5,0($10)	10
5	lw $2,65524($10)	11
6	addi $5,$2,4	14
7	bne $5,$2,$l5	17
8	add $0,$0,$0	18

Total de cycles : 18

RENOMMAGE

Instruction	Code	Cycle de sortie
0	lw $9,0($6)	1
1	lw $2,0($9)	4
2	add $11,$14,$2	7
3	ori $10,$6,0	8
4	sw $11,0($10)	10
5	lw $2,65524($10)	11
6	addi $5,$2,4	14
7	bne $5,$2,$l5	17
8	add $0,$0,$0	18

Total de cycles : 18

SCHEDULING

Instruction	Code	Cycle de sortie
0	lw $4,0($6)	1
3	ori $10,$6,0	7
1	lw $2,0($4)	2
2	add $5,$14,$2	4
5	lw $2,65524($10)	9
4	sw $5,0($10)	8
6	addi $5,$2,4	11
7	bne $5,$2,$l5	14
8	add $0,$0,$0	15

Total de cycles : 15

RENOMMAGE & SCHEDULING

Instruction	Code	Cycle de sortie
0	lw $9,0($6)	1
3	ori $10,$6,0	7
1	lw $2,0($9)	2
2	add $11,$14,$2	4
5	lw $2,65524($10)	9
4	sw $11,0($10)	8
6	addi $5,$2,4	11
7	bne $5,$2,$l5	14
8	add $0,$0,$0	15

Total de cycles : 15

NOTE IMPORTANTE :

Par rapport au renommage fait lors du TD, on constate que reg_rename() n'a pas renommé le registre $2 dans sa définition à i1, et dans ses utilisations ultérieures (i2).

En effet, le registre $2 est en fait vivant en sortie du bloc 0 (puisque BB0 a pour successeurs BB1 et BB3 et que BB3 a $2 dans son Live-In).

Le registre $2 ne peut donc pas être renommé dans BB0.

Donc la dépendance WAR de i2 à i5 ne peut pas être évitée.

Cela explique la différence d'optimisation trouvée en TD (nb_cycles = 9 à la fin de BB0) et celle trouvée par notre projet (nb_cycles = 11 à la fin de BB0)

Conclusion

Nous avons implémenté avec succès toute les fonctions de traitement du code ce qui nous a permis d'apporter des optimisations quant au nombre de cycles nécessaire à l'exécution d'un programme.

Les points les plus importants que nous retenons de cette réalisation sont :

- la nécessité de cartographier les blocs de bases et leurs relations
- la primordialité de l'identification des dépendances entre les instructions
- la reconnaissance des plages d'utilisation des registres ainsi que la durée d'exécution des instructions

Grâce à ce projet nous avons approfondi notre compréhension des opérations se passant à bas niveau et d'autant mieux intégré les enjeux liés à la compilation.