Rapport
Projet de Compilation Avancée

Amel ARKOUB 3301571
Ling-Chun SO 3414546

20 mai 2018

Sommaire

0

Présentation

0.1 Introduction

Le projet de Compilation Avancée consiste en l'étude de code MIPS32 dans l'optique d'optimiser le temps d'exécution en nombre de cycles, notamment par ré-ordonnancement et renommage de variables. Nous devons modéliser le code brut en une structure de données de manière à pouvoir l'analyser efficacement et d'y appliquer des algorithmes d'analyses et de transformation de code.

0.2 Avancement

Voici la liste exhaustive des fonctions implantées et de leur avancement :

- Function::compute_basic_block() → fonctionnelle
- Function::comput_succ_pred_BB() → fonctionnelle
- Program::comput_CFG() → fonctionnelle
- Function::compute_dom() → fonctionnelle
- Basic_Block::compute_pred_succ() → fonctionnelle
- Basic_Block::nb_cycle() → fonctionnelle
- Basic_Block::compute_use_def() → fonctionnelle
- Function::compute_live_var() → fonctionnelle
- Basic_Block::compute_rename(list<int>*) → fonctionnelle
- Basic_Block::compute_rename() → fonctionnelle

0.3 Fichier de tests

Voici une description succincte des exécutables :

- main → teste toutes les fonctions sauf les renames. Notez que le calcul des registres vivants en début ou fin de bloc est décrit à la fin de l'exécution, de même pour les registres définis vivants en sortie. Aussi, il ne faut pas considérer les LiveIn et LiveOut qui ne sont pas à la fin car il faut calculer les Uses et Defs avant de calculer les LiveIn et LiveOut, ce qui n'est fait qu'à la fin du code.

- test_rename → teste le renommage de variables.
- test_rename_list → test le renommage de variables à partir d'une liste.

1

Fonctions implantées

1.1 Function::compute_basic_block()

La fonction Function::compute_basic_block() permet de calculer tous les blocs basiques d'une fonction. Il s'agit d'un algorithme qui parcourt toutes les instructions et qui calcule le début et la fin d'un bloc selon certaines règles, celles-ci étant explicitées ci-dessous.

```
debut <- _head->get_next()
fin <- vide
k <- 0
current <- première instruction qui n'est pas une directive
Tant current n'est pas l'instruction après la dernière
    si current est une instruction alors
        si debut est vide alors
            debut <- current
        si current est un branchement alors
            current <- current->get_next()
            add_BB(debut, current, current->get_prev(), k++) //car il y a le delayed
            debut <- vide

    si current est un label alors
        si debut n'est pas vide alors
             add_BB(debut, current, NULL, k++)
        debut = current

    current <- current->get_next()

Fin tant que
```

1.2 Function::comput_succ_pred_BB()

La fonction Function:comput_succ_pred_BB() calcule pour chaque bloc ses blocs successeurs et son bloc prédécesseur.

L'algorithme s'organise comme ceci : pour chaque bloc, on récupère l'instruction de branchement qu'il contient, si celle-ci existe. Si cette instruction est un branchement conditionnel, cela signifie

qu'elle possède deux successeurs : le bloc suivant et le bloc labellé de la condition. Sinon, si cette instruction est un appel de fonction, alors son seul bloc successeur est le suivant. Sinon, si c'est un branchement incontionnel, son seul bloc successeur est le bloc labellé par le branchement. Si cette instruction n'existe pas, son seul bloc successeur est le suivant.
Il faut noter qu'à chaque fois qu'on associe un successeur au bloc, le successeur se voit attribuer le bloc courant comme prédécesseur.

1.3 Program::comput_CFG()

Cette fonction correspond à un simple parcours de toutes les fonctions (structure de données) du programme et créer un CFG associés, ces CFG sont ensuite ajoutés dans une liste qui contiendra donc toutes les CFG.

1.4 Function::compute_dom()

Function::compute_dom() permet de calculer les dominances entre blocs. Commençons d'abord par décrire les structures de données manipulées au cours de l'algorithme.

- Une liste blocs workinglist contenant les blocs qui doivent être analysés
- Un booléen change signalant si un changement concernant les blocs qui le domine a été effectué

L'algorithme s'organise de la manière suivante.

On commence par ajouter tous les blocs sans prédécesseur dans la liste workinglist.

Tant que cette liste n'est pas vide, on ôte chaque bloc contenu pour l'analyser. Notons que le booléen change vaut False à chaque début d'analyse.

Si ce bloc a un seul prédécesseur, cela signifie qu'il est dominé par lui. Si cette information n'avait pas déjà été ajoutée au bloc, alors on passe le booléen change à True.

Sinon, si ce bloc a plusieurs prédécesseurs, on commence par récupérer tous les blocs dominants en commun de tous les prédécesseurs de ce bloc, car ces dominants communs dominent le bloc. Ensuite on regarde si un ou plusieurs des prédécesseurs du bloc dominent tous les autres prédécesseurs du bloc. Le cas échéant, ils dominent alors le bloc. Comme précédemment, si on a ajouté au moins prédécesseur comme dominant du bloc, alors on passe le booléen change à True.

Si le booléen change vaut True, c'est-à-dire s'il y a eu au moins un changement quant à l'ajout des dominants du bloc, alors on doit réévaluer tous les successeurs du bloc. Ainsi, on les ajoute tous à worklist.

L'algorithme s'arrête quand la worklist est vide.

1.5 Basic_Block::compute_pred_succ()

La fonction Basic_Block::compute_pred_succ() calcule toutes les dépendances entre les instructions du bloc. Pour ce faire, on dispose des tableaux et des listes suivants :

- Le tableau rawTab, de taille 64, dont l'indice k correspond au numéro du regitre et donc la case rawTab[k] représente la dernière écriture effectuée par l'instruction i. Ainsi, par exemple, suite à l'instruction i6 : addiu $2, $4, 1, rawTab[2]=6.

- C'est sur le même raisonnement qu'on utilise le tableau warTab, de taille 64, à l'exception que la case d'indice k contient la liste des dernières instructions ayant lu un même registre. Par exemple, considérons la suite d'instructions suivante :
  i3 : addiu $4, $6, 8
  i4 : ori $5, $6, $7
  i5 : add $9, $10, $6
  Alors warTab[6] = 3, 4, 5

- La liste d'instructions lastMemInst est la liste des instructions mémoire déjà rencontrées. Cette liste est utile pour qu'à la prochaine instruction mémoire, on puisse vérifier s'il existe une dépendance MEM entre elle et toutes celles de la liste.

- Le tableau hasDep de taille du nombre d'instructions du bloc. Si une instruction n'a aucune dépendance, alors on établit une dépendance CONTROL à l'instruction de branchement finale.

Passons à l'explication générale de l'algorithme.

Pour chaque instruction, si celle-ci a un registre source1 (et/ou un registre source2) dont le numéro est k (k compris entre 0 et 63), on regarde dans rawTab[k] la dernière instruction qui a écrit dans le registre (si elle est existe). Le cas échéant, on établit une dépendance RAW entre l'instruction rawTab[k] et celle-ci.
Pour chaque registre source k que possède l'instruction, on ajoute le numéro de l'instruction à la liste warTab[k], car il s'agit là d'une lecture effectuée sur ce registre.

Si l'instruction possède un registre de destination de numéro j, on établit une dépendance WAR entre elle et chaque instruction contenue dans la liste à la case warTab[j]. On vide cette liste.
De plus, si rawTab[j]=n (n est un numéro d'instruction), alors on établit une dépendace WAW entre l'instruction et n.

Si cette instruction est une instruction mémoire, alors on regarde si cette instruction et toutes les autres instructions mémoire déjà rencontrées auparavant, contenues dans la liste lastMemInst, on une dépendance MEM. Le cas échéant, on établit cette dépendance.

Finalement, une fois qu'on a ajouté les dépendances existante à toutes les instructions, on regarde celles qui n'en ont aucune. Alors on ajoute une dépendance de CONTROL entre elles et la dernière instruction de branchement finale.

1.6 Basic_Block::nb_cycle()

Le calcul du temps de cycle d'un bloc est calculé par la formule suivante:
cycle(0) = 1 (il n'a pas de dépendance avec un prédécesseur et on a supposé qu'à chaque cycle une instruction sort du pipeline)
Soit i, l'indice de l'instruction et p $\in$ Predecesseurss(i).
cycle(i) = maxcycle(i-1) + 1, maxcycle(p) + delai(p,i) avec i $>$ 0

Ainsi l'algorithme correspond à un double parcours, la première itère sur toutes les instructions pour calculer chaque cycle de sortie et la seconde sur les prédécesseurs (au sens de dépen-

dance) de chaque instruction pour calculer le plus grand cycle possible pour l'instruction à considérer.

1.7 Basic_Block::compute_use_def()

Cette fonction calcul les registres utilisés et les registres définit à l'intérieur d'un bloc de base.

```
On initialise deux tableaux de booléen de taille NB_REG (Use et Def)
Pour toutes instructions inst dans le BB
    Si il existe un registre source src1 et
    qui est n'est pas définit dans Def alors
        il est utilisé (Use[src1] = true)
    Si il existe un registre source src2 et
    qui est n'est pas définit dans Def alors
        il est utilisé (Use[src2] = true)
    Si il existe un registre destination dst alors
        le registre est définit (Def[dst] = true)
Fin pour tout
Si l'avant dernière instruction est un "jal" alors
    les registre 4,5,6 sont utilisés (Use[4] = Use[5] = Use[6] = true)
    les registres 2 et 29 sont définis (Def[2] = Def[29] = true)
```

1.8 Function::compute_live_var()

La fonction analyse les registres vivants en entrée et en sortie des blocs de base. Elle consiste à la recherche des blocs sans successeurs et au calcul d'une formule, si le résultat est différent de celui original alors on ajoute ses prédécesseurs, puisque la formule utilise les blocs successeurs donc il faut recalculer de nouveau.
La formule est la suivante:
LiveOut(BB) = LiveIn(P) avec P $\in$ successeurs(BB)
LiveIn(BB) = Use(BB) $\cup$ {LiveOut(BB) $\setminus$ DEF(BB)}

```
workinglist <- vide (Basic Block)
change <- false
On recherche les blocs sans successeurs et on l'ajoute dans la workinglist
Tant que la workinglist n'est pas vide
    bb <- workinglist.pop(0)
    //on applique la formule LiveOut suivant le nombre de successeurs
    switch(bb->nb_succ()) //il y a au maximum 2 successeurs avec les jumps
        case 0: //c'est le cas où il n'y a pas de successeurs
            bb->LiveOut[2] = true
            bb->LiveOut[29] = true
        case 1:
            pour tout i dans NB_REG
                bb->LiveOut[i] = bb->get_successor1()[i]
            fin pour tout
        case 2:
            pour tout i dans NB_REG
                bb->LiveOut[i] = bb->get_successor1()[i]
            fin pour tout
```

```
            pour tout i dans NB_REG
                si  bb->LiveOut[i] = true alors
                    bb->LiveOut[i] = true
            fin pour tout

    si il y a un changement entre l'ancien LiveOut et celui calculé alors
        change <- true

    //on applique la formule de LiveIn
    pour tout i dans NB_REG
        bb->LiveIn[i] = bb->Use[i]
        si bb->LiveOut[i] = true et bb->Def[i] = false alors
            bb->LiveIn[i] = true
    fin pour tout

     si il y a un changement entre l'ancien LiveIn et celui calculé alors
        change <- true

    si change = true
        On ajoute tout les predecesseurs de bb dans la workinglist

Fin tant que
```

1.9 Basic_Block::compute_def_liveout()

Cette fonction calcul les registres définits et vivants en sortie de bloc et indique à quelle instruction elle a été définit (la dernière si plusieurs définitions). Cela correspond à un simple parcours de toutes les instructions, si l'instruction possède un registre de destination dst alors c'est une définition, de plus si LiveOut[dst] = true alors on ajoute l'indice de l'instruction dans DefLiveOut[dst].

1.10 Basic_Block::compute_rename(list<int>*)

Cette fonction permet de renommer les registres qui peuvent l'être sans remettre en cause la cohérence du code.

```
On parcours toutes les instructions du BB
    Si l'instruction possède un registre de destination dst alors
        Si ce registre n'est pas vivante en sortie ou
        du moins n'est pas la dernière définition en sortie alors
            newr <- un registre libre de la liste
            On cherche toute les dépendances RAW avec cette instructions
                on renomme dans ces instructions le registre src = dst par newr
            On remplace le registre dst par le registre newr
```

1.11 Basic_Block::compute_rename()

Cette fonction renomme de la même manière que la fonction précédente mais qui de plus trouve les registres libres pour renommer. Cela correspond à un simple parcours, tout registres i qui verifie i $\notin$ {0, 26, 27, 28, 29, 30, 31} and !LiveIn[i] and !Def[i] est un registre utilisable pour renommer, il suffit donc d'ajouter dans une liste et appeler la fonction précédente, Basic_Block::compute_rename(list<int>*).

2

Analyse

nb_cycle/fichier	origine	réordonnancement	renommage	renommage + réordonnancement
dep_inst3.s/f0bb0	13	12	11	11
dep_inst3.s/f0bb1	10	10	10	10
dep_inst3.s/f0bb2	6	5	5	5
dep_inst3.s/f0bb3	9	9	9	9
aes_00.s/f0bb0	12	12	11	11
aes_00.s/f0bb1	14	14	14	14
aes_00.s/f0bb2	5	5	5	5
aes_00.s/f0bb3	8	9	7	7
aes_00.s/f1bb0	12	12	11	11
aes_00.s/f1bb1	17	17	17	13
aes_00.s/f1bb2	13	11	11	11
aes_00.s/f1bb3	10	11	10	10
aes_00.s/f2bb0	12	12	11	11
aes_00.s/f2bb1	17	17	17	13
aes_00.s/f2bb2	13	11	11	11
aes_00.s/f2bb3	10	11	10	10
aes_00.s/f3bb0	13	13	12	12
aes_00.s/f3bb1	18	19	18	14
aes_00.s/f3bb2	13	11	11	11
aes_00.s/f3bb3	10	11	10	10
aes_00.s/f4bb0	14	14	13	13
aes_00.s/f4bb1	46	46	46	34
aes_00.s/f4bb2	13	11	11	11
aes_00.s/f4bb3	10	11	10	10
aes_00.s/f5bb0	148	140	140	130
aes_00.s/f7bb1	52	52	48	39
aes_00.s/f7bb2	26	26	25	21
aes_00.s/f8bb1	41	41	37	30
aes_00.s/f8bb6	18	18	17	15
aes_00.s/f8bb7	26	26	25	21

Nous avons lancés les algorithmes de renommage et de réordonnancement et analysés le temps d'éxecution ci-dessus. Nous nous sommes basés sur deux types de code, un code court (dep_inst3.s) et un code beaucoup plus long (aes_00.s) afin d'observer la variation suivant la taille du code.

Nous pouvons observer dans les résultats du fichier de dep_inst3.s qu'il y a des cas de gains de cycles en réordonnançant ou lorsqu'il y a renommage. Mais dans la moitié des cas il n'y aucun gain, aucune perte.
Cependant, dans les résultats du fichier aes_00.s, nous pouvons observer qu'il y a des cas de perte en temps de cycle lorsqu'il y n'y a uniquement que du réordonnancemennt. A contrario nous n'observons pour le renommage seulement au pire aucun gains d'autant que les gains s'il y en a, sont meilleurs que celui du réordonnancement seul.
Malgré tout nous pouvons observer de très bonne performance avec le couplage du renommage suivi de réordonnancement, les gains peuvent jusqu'à doubler en terme de gains de cycles.
Par exemple, aes_00.s/f7bb1 nous passons d'un nombre de cycle de 52 à 39, soit un gain de 25%. De manière générale nous pouvons observer que les codes court (10 instructions) ont un gains $< 10\%$, les codes moyens (40-50 instructions) ont un gains de 25% et les codes longs ($>$ 100 instructions) ont un gains de 10%.

3

Conclusion

L'analyse et l'étude du code assembleur MIPS32 nous a permis d'optimiser le temps d'éxecution en nombre de cycle du programme. Nous avons pu observer expérimentalement qu'il est très souhaitable de coupler le renommage et le réordonnancement plutôt que de les séparer, les gains ne sont pas négligeables. Ces types d'analyses ont tout interêt à être implantés dans les compilateurs tant le gains est non négligeables et le temps d'éxecution relativement faible (par exemple le aes_00.s contient 2000 lignes de codes et s'exécute en un faible délai).