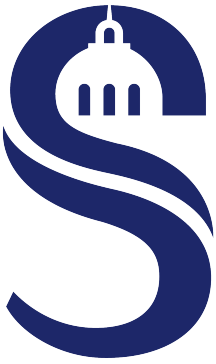

SORBONNE UNIVERSITÉ

Master Informatique

Rapport de projet
Compilation avancée

Adan Bougherara
Vivien Demeulenaere

Remis le 22 Avril 2022

Table des matières

1 Les fonctions implantées

Toutes les méthodes présentées ci-dessous sont fonctionnelles avec les tests fournis. Tous les codes générés s'exécutent sur MARS.
Les méthodes `compute_basic_block`, `compute_succ_pred_BB` et `compute_dom` appartiennent à la classe `Function`.
Les méthodes A, B et C appartiennent à la classe `Basic_block`

1.1 compute_basic_block

En vue d'effectuer certaines optimisations sur du code assembleur, il parait judicieux de travailler par blocs de base. La méthode `compute_basic_block` permet de calculer ces blocs. Pour se faire, un découpage du code est effectué dès que l'on rencontre une étiquette ou un branchement. On prendra soin de conserver le *delayed slot* à la suite du branchement. Le *delayed slot* est l'instruction suivant un branchement qui est exécutée même si l'on effectue un saut. Dans le cas de notre processeur, il n'y a qu'une seule instruction lue après une instruction de branchement.

1.2 compute_succ_pred_BB

Le calcul des blocs succédant et précédant un bloc de base est effectué par la méthode `compute_succ_pred_BB`. Cette méthode parcourt les blocs et ajoute des liens entre eux (*i.e.* on trouve les successeurs du bloc courant sans oublier d'ajouter ce dernier comme prédécesseur de ses successeurs). Quatre cas principaux sont alors à distinguer :

- Si le bloc se termine par un branchement inconditionnel, alors un lien est créé entre le bloc courant et le bloc cible du saut.
- Si le bloc se termine par un branchement conditionnel, alors un lien est ajouté entre le bloc courant et le suivant. Un deuxième lien est également ajouté entre le bloc courant et le bloc ciblé par le saut.
- Si le bloc courant est la fin d'une fonction, alors aucune dépendance n'est ajouté.
- Sinon, un lien est ajouté entre le bloc courant et le suivant.

1.3 compute_dom

La méthode `compute_dom` calcule les blocs de base dominants des blocs de base d'une fonction.
Notons `Domin` le bitset associé à chaque bloc et `T` un bitset temporaire. `Domin` permet d'indiquer si le bloc est dominé ou non par un autre bloc de la fonction.
La fonction initialise une liste de travail contenant au départ le bloc d'entrée de la fonction, qui ne possède pas de prédécesseur.
Tant que la liste de travail n'est pas vide, on parcourt les blocs contenus dans cette liste.
S'il s'agit du bloc racine, on met à 0 `T`, pour indiquer qu'il n'est dominé par aucun bloc.
Sinon, on met des 1 dans `T`, pour indiquer qu'il est dominé par tous les blocs.
Puis, on met à jour `T` comme étant l'intersection des `Domin` de ses blocs prédecesseurs.
Deux cas sont alors possibles pour le bloc :

- Son `Domin` est égal `T`. On enlève alors le bloc de la liste de travail.
- Son `Domin` est différent `T`. On enlève alors le bloc de la liste de travail et on y ajoute ses successeurs.

1.4 compute_loops

La méthode détermine les boucles dans une fonction et les blocs de base composant chaque boucle. Une boucle est représenté par la classe `Loop`. Son constructeur a besoin de l'entête et du bloc source de l'arc retour. Pour les trouver, `compute_loops` parcourt les blocs de la fonction. Si un bloc est dominé par un de ses successeurs, une boucle est alors détectée, l'entête correspondant au successeur et le bloc source correspondant au bloc courant.
Enfin, pour déterminer quels blocs sont impliqués dans la boucle, elle fait appel à une méthode de la classse `Loop`, `compute_in_loop_BB`.
Cette dernière parcourt les blocs en partant de l'entête et en ajoutant petit à petit les prédécesseurs dans la `Loop` jusqu'à retomber sur le bloc source.

1.5 comput_pred_succ

Afin de gagner des cycles d'exécution pour un bloc de base, on cherche à réordonnancer les instructions pour éviter un maximum les cycles de gel. Il est donc nécessaire de calculer les dépendances entre instructions pour savoir si l'on peut en changer l'ordre d'exécution. Ce travail est effectué dans la méthode `comput_pred_succ`. Celle-ci parcourt en arrière le bloc de base. Elle ajoute alors les dépendances suivantes :

- Une dépendance RAW si l'instruction courante lit dans un registre après qu'il ait été écrit et qu'aucune dépendance RAW pour le registre incriminé n'ait déjà été ajoutée.
- Une dépendance WAW si l'instruction courante lit dans un registre après qu'il ait été lu et qu'aucune dépendance WAW n'ait déjà été ajoutée.
- Une dépendance WAR si l'instruction courante écrit dans un registre après qu'il ait été lu et qu'aucune dépendance WAW n'est déjà été ajoutée.
- Une dépendance MEM si l'instruction courante accède à un emplacement mémoire modifié par une autre instruction. Aucune dépendance mémoire ne sera ajoutée si les instruction lisent le même registre avec un offset différant d'au moins quatre pour accéder à la mémoire et que le registre en question n'a pas été modifié entre temps.

Il est à noté que nous avions dans un premier temps adopté une approche assez stricte concernant les dépendances mémoires. En effet, les dépendances étaient systématiquement ajoutées même si la lecture et l'écriture ne se faisaient pas au même emplacement mémoire, bridant ainsi les possibilités de ré-ordonnancement du code. C'est pour cela qu'il a par la suite été choisi de prêter attention aux registres en jeu et de ne pas ajouter de dépendance si ce n'était pas nécessaire.

1.6 nb_cycles

Afin de quantifier les différents gains que l'on obtient lors de diverses optimisations, il est impératif de pourvoir évaluer le nombre de cycles nécessaires à un bloc pour s'exécuter. C'est donc ce que la méthode `nb_cycles` vise à calculer. Pour cela, on supposera qu'une instruction peut sortir du pipeline à chaque cycle. Ainsi, sans cycle de gel, une instruction prendra un cycle pour s'exécuter. On ajoutera de plus la durée des cycles de gel à partir dépendances précédemment calculées et d'une matrice codée en dur contenant le temps de gel induit par les diverses dépendances.

1.7 compute_use_def

La méthode calcule les ensembles des registres définis et utilisés, nécessaire à l'analyse des registres vivants.
Notons `Def` et `Use` les bitset des registres respectivement définis et utilisés. `Def` et `Use` sont tous deux initialisés avec des 0. Notons i une instruction.
On parcourt les instructions du bloc en partant du début et on suit la procédure suivante :
Si i lit dans un registre, de numéro Ri, et que Ri n'a pas été écrit par une instruction précédente du bloc, alors on met à jour `Use[Ri]` à 1.
Si i écrit dans un registre, de numéro Ri, alors on met à jour `Def[Ri]` à 1.

On peut noter qu'il est important de tester les registres sources avant le registre destination pour éviter des erreurs. En effet, tester l'écriture avant les lectures pose problème dans le cas ou une instruction écrit et utilise un même registre car ce dernier ne sera pas considéré comme utilisé par le bloc.
Dans nos tests, cette erreur s'est révélée lors de l'exécution du fichier `aes_ca.s` en modifiant la valeur de retour attendue 0 par 1.

1.8 compute_live_var

La méthode calcule les registres vivants en entrée et en sortie d'un bloc. L'ensemble des variables vivantes en entrée et en sortie d'un bloc sont respectivement représentés dans le code par les bitset $LiveIn$ et $LiveOut$. Cette méthode fonctionne en deux temps.

Premièrement, il s'agit d'initialiser notre liste de travail ainsi que le $LiveOut$ des blocs de sortie et ceux terminant par des appels :

- Pour un bloc sans successeur, ce qui correspond uniquement au bloc de sortie d'une fonction ou du main dans ce projet, on indique que les registres R2, R29 ainsi que les registres de R16 à R23 sont vivants en sortie du bloc. De plus, on ajoute le bloc à la liste de travail.
- Pour un bloc se terminant par un appel de fonction (suivi du delayed slot), on met à jour le $LiveOut$ du bloc en indiquant que R4, R5, R6 et R7 sont vivants en sortie du bloc.
- Pour un bloc se terminant par un appel système (suivi du delayed slot), on met à jour le $LiveOut$ du bloc en indiquant que R2, R4, R5, R6 et R7 sont vivants en sortie du bloc.

Dans un second temps, on parcourt les blocs de notre liste de travail tant qu'elle n'est pas vide. On retire le bloc courant de la liste de travail. On définit le $LiveOut$ d'un bloc comme étant l'union des $LiveIn$ de tous ses successeurs. On définit le $LiveIn$ d'un bloc comme étant l'union des registres utilisés par ce bloc (Use) et des registres de $LiveOut$ du bloc moins les registres définis (Def) par ce bloc.
Si lors de l'itération courante, le $LiveIn$ du bloc a été modifié, alors on ajoute tous les prédécesseurs du bloc dans la liste de travail afin de mettre à jour leurs variables vivantes en entrée et en sortie suite aux modifications apportées au bloc courant.

1.9 compute_def_liveout

La méthode calcule les index des instructions d'un bloc qui définissent un registre vivant en sortie du bloc. Il s'agit ici de mettre à jour le tableau `DefLiveOut` du bloc, initialisé avec -1 pour chaque élément. Notons i une instruction et $index\_i$ son indice dans le bloc.
On parcourt les instructions du bloc en partant du début et on suit la procédure suivante :
Si i écrit dans un registre, de numéro Ri, et que le registre Ri est vivant en sortie du bloc, c'est-à-dire que le bitset `LiveOut` du bloc vaut 1 à l'indice Ri, alors on met à jour le champ correspondant du tableau : `DefLiveOut[`Ri`] =` $index\_i$.

1.10 reg_rename

La méthode `reg_rename` sans paramètre initialise une liste des registres renommables et appelle `reg_rename` avec cette liste en paramètre. Cette liste contient tous les registres à l'exception des registres $R0$, $R2$, $R29$, $R31$ ainsi que tous les registres définis dans le bloc ou vivants en entrée du bloc.
Notons i une instruction du bloc.
On parcourt les instructions du bloc en partant du début et on suit la procédure suivante :
Si la liste des registres libres est vide, on arrête le renommage. Si i écrit dans un registre, de numéro Ri, et que cette définition du registre n'est pas vivante en sortie du bloc, alors le registre est renommable (A noter qu'on ne renomme pas les registres R29 et R31). On renomme alors le registre dans i et on propage le renommage vers les instructions ayant une dépendance RAW avec i. Il s'agit alors de changer Ri dans les sources par le nouveau registre.

2 Analyse des effets du renommage et du réordonnancement

Afin de comparer les effets des diverses opérations dans le code, on utilisera deux dictionnaires de délai différents notés `délai 1` et `délai 2` et présents dans le fichier `Enum_type.h`. Dans le premier dictionnaire, les délais sont plus courts que dans le deuxième.
Ainsi, pour chacun des fichiers fournis, on compte le nombre total de cycles nécessaires à l'exécution du programme. On répète ensuite l'opération en effectuant un renommage, un réordonnancement ou encore un renommage puis un réordonnancement.

Fichier	Code d'origine	renommé	réordonnancé	renommé puis réordonnancé
`max_fct.s`	158	158	143	140
`mat_mul.s`	407	407	361	339
`tab_fct.s`	417	417	352	336
`aes_ca.s`	1937	1937	1665	1576
Gain moyen (en %)	$\emptyset$	0	12.6	16.5

Table 1 – Nombre de cycle d'exécution avec `délai 1`

Fichier	Code d'origine	renommé	réordonnancé	renommé puis réordonnancé
`max_fct.s`	192	192	174	171
`mat_mul.s`	547	547	495	404
`tab_fct.s`	546	546	466	396
`aes_ca.s`	2712	2712	2252	1828
Gain moyen (en %)	$\emptyset$	0	12.6	24.2

Table 2 – Nombre de cycle d'exécution avec `délai 2`

On peut constater que le réordonnancement des instructions nous apporte un gain moyen de 12.6 % avec les deux délais. Ce résultat paraît cohérent puisqu'en réordonnançant on se contente de supprimer le plus possible les cycles de gel.
En ce qui concerne le renommage seul, aucun gain n'est observé puisqu'il permet d'enlever les dépendances de nom (WAW et WAR) toutefois le code reste exécuté dans le même ordre.
En revanche associer le renommage au réordonnancement permet un gain bien plus important avec en moyenne 24 % de gain sur le `délai 2`. Ceci s'explique par le fait que le réordonnancement s'effectue sur un code privé des dépendances de nom. Il a donc moins de contraintes pour réarranger l'ordre des instructions.

3 Question ouverte

En effectuant des tests de renommage sur les codes provenant des fichiers `mat_mul.s` et `aes_ca.s`, nous avons remarqué que la liste des registres utilisables pour le renommage s'épuise très vite sur les gros blocs, ce qui a pour effet de ne pas renommer une grande partie du bloc et par conséquent de ne pas profiter pleinement de l'effet du renommage pour réduire le nombre de cycles lors de la phase de scheduling qui suit.
Afin d'éviter que cette liste soit vide, nous avons modifié l'algorithme de renommage. Lorsque l'on récupère un registre de la liste, on effectue le traitement habituel (on effectue le renommage sur l'instruction i puis on propage sur les instructions en dépendance RAW avec i), puis on ré-ajoute le registre en fin de notre liste de registres utilisables.
Ainsi, on gagne beaucoup de cycles sur des codes avec de gros blocs.
Avec le fichier `aes_ca.s`, en effectuant un renommage puis un réordonnancement sur `délai 1` (`test_renommage_scheduling.cpp`), on passe de 326 cycles gagnés à 361. Concernant `mat_mul.s`, on passe de 56 cycles gagnés à 68.
On peut noter que sur les petits blocs où la liste ne se vide pas, aucun gain supplémentaire est apporté.

4 Conclusion

La réalisation de toutes les étapes de ce projet nous a permit de nous rendre compte de l'importance cruciale de l'optimisation du code lors de la phase de compilation. En effet, nous avons pu observer sur des fichiers relativement réduits des résultats déjà conséquents, ce qui laisse présager un gain considérable de temps d'exécution sur de très longs programmes.
Ce projet, nous a également permit de découvrir le langage `c++` assez largement utilisé dans le monde du travail. De plus, il a été assez agréable de pouvoir travailler sur le projet lors des heures de travaux pratiques afin de bénéficier de votre aide.