Projet CPA
Vectorisation de boucle

Pierre Aubert, Salim Nahi, Puchen Liu

16 novembre 2014

Introduction

Le but de ce projet est de fournir un plugin GCC et une librairie permettant d'étudier les accès mémoire du programme et ainsi déterminer si certaines boucles sont vectorisables ou non.

Table des matières

1 Analyse du problème et architecture du projet

Comme les plugins GCC 4.9 s'écrivent en C++, nous avons choisi le langage C++. Nous avons séparé le projet en deux grandes parties. La première est un plugin GCC qui permet d'ajouter des appels de fonction pour la deuxième partie qui elle est une librairies qui analyse les accès mémoires. Pour permettre une avancée simultannée des deux parties nous avons ajouté des fonctions en C qui permettent d'appeler la librairie manuellement et ce sans utiliser le plugin, et permettait aussi de tester le plugin sans avoir à utiliser la librairie. Nous avons également dévoloppé plusieurs tests afin de vérifier le bon fonctionnement des parties développées.

2 L'analyse dynamique

L'analyse dynamique conciste à rajouter des appels de fonction à la librairie d'analyse lors de la compilation. Ces fonctions seront appelées durant l'exécution du programme à analyser.

3 Le plugin

Le plugin permet, dans un premier temps, d'insérer les appels à la librarie d'analyse lors de la compilation puis, dans un second temps d'effectuer une analyse statique des boucles les plus internes afin de déterminer avant l'exécution si elles sont vectorisables, ou non.

3.1 Spécification d'une fonction à analyser

Pour spécifier une fonction à prendre en compte avec le plugin, il faut ajouter un pragma de la forme suivante :

```
#pragma mihp vcheck ...
```

Il est possible de préciser une fonction à analyser :

```
#pragma mihp vcheck fonction
```

Ou une liste de fonction à analyser :

```
#pragma mihp vcheck (func1, func2, func3, ..., funcN)
```

3.2 Analyse d'une boucle

Dans ce projet, nous ne traiterons que les boucles les plus internes. Pour ce faire, nous analyserons chaque fonction spécifiée dans le **pragma mihp vcheck**. L'analyse conciste dans un premier temps à vérifier que la fonction spécifiée contient bien une boucle, une fois que la boucle la plus interne à été déterminée, le plugin ajoute différents appels à la librairie :

- Une fonction qui spécifie le début d'une boucle
- Une fonction qui spécifie la fin d'une boucle
- Une fonction qui spécifie une nouvelle iteration
- Une fonction qui spécifie un accès mémoire

Le début de la boucle est précisé par la fonction **mihp_newLoop** , cette fonction sera appelée juste avant le **header** de la boucle. La fonction **mihp_endLoop** spécifie la fin d'une boucle et sera appelée sur chaque arrête sortante de la boucle. La fonction **mihp_newIteration** indique une nouvelle itération, et sera appelée juste avant le **latch** de la boucle. La fonction **mihp_address** quant à elle, sera appelée pour chaque opérande pertinente d'un **statement gimple** du type **GIMPLE_ASSIGN**. Nous expliquerons ce qu'est une opérande pertinente un peut plus bas.

Les positions des différents appels aux fonctions de la biliothèque d'analyse sont illustré dans la figure 1.

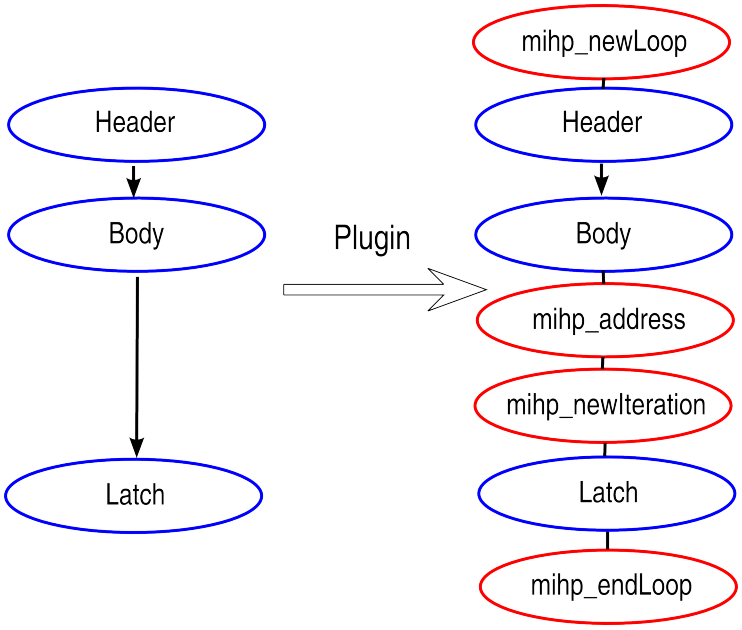

FIGURE 1 – Illustration de l'ajout des appels au fonctions de la bibliothèque d'analyse par le plugin.

3.2.1 Analyse naïve

Dans un premier temps nous avons pris en compte toutes les opérandes non constantes, puisque les opérandes constantes ne peuvent être que lues (ce qui ne pose aucun problème pour la vectorisation). Comme gimple converti toutes les expressions en diadiques ou en triadique, il créé des variables temporaires de la forme **D.XXXX**, les adresses de ces variables sont généralement **0x1**, **0x2**, **0x3**, ect. Nous avons donc fait des tests pour vérifier que cela ne posait pas de problème avec notre librairie d'analyse (qui sera détaillée dans la section suivante).

Nous avons vite mis en évidence le fait que les variables temporaires créées par Gimple allaient nous poser problème, car elles ont toujours les mêmes adresses. La méthode naïve, consiste à tenir uniquement compte des varaibles décrites par **PARM_DECL**, ce qui nous permet de n'utiliser que les variables définies dans le fichier source. Le problème est que ces varaibles sont toujours accédées en lecture. Il n'est donc pas possible de se contenter uniquement de ces varaibles.

3.2.2 Analyse efficace des opérandes

Comme nous l'avons vu dans la section précédente, l'anayse des opérandes ne peut se limité aux varaibles définies par l'utilisateur dans le fichier source. Nous devons donc analyser les opérations que Gimple génère à la compilation.

Si nous prenons une itération simple, comme celle ci-dessous :

```
c[i] = a[i] + b[i];
```

Il est naturel d'imaginer les appels de fonctions suivants :

```
mihp_address(&b[i], sizeof(b[i]), MIHP_READ);
mihp_address(&a[i], sizeof(a[i]), MIHP_READ);
mihp_address(&c[i], sizeof(c[i]), MIHP_WRITE);
```

Mais si nous observons, ce que génère Gimple nous constatons que les variables pertinentes sont temporaires de la forme **D.XXXX** :

```
D.2741 = (long unsigned int) i;
D.2742 = D.2741 * 4;
D.2743 = c + D.2742;                    => &c[i]
D.2744 = (long unsigned int) i;
```

```
D.2745 = D.2744 * 4;
D.2746 = a + D.2745;                    => &a[i]
D.2747 = *D.2746;                       => a[i]
D.2748 = (long unsigned int) i;
D.2749 = D.2748 * 4;
D.2750 = b + D.2749;                    => &b[i]
D.2751 = *D.2750;                       => b[i]
D.2752 = D.2747 + D.2751;               => tmp = a[i] + b[i]
*D.2743 = D.2752;                       => c[i] = tmp
```

On remarque dans cet exemple que si une varaible **PARM_DECL** n'est pas l'opérande la plus a gauche, alors on lit d'adresse qui est dans l'opérande de gauche. Et si l'opérande de gauche est un **MEM_REF**, cette adresse sera alors écrite.

Le plugin a donc besoin de deux fonctions : une qui permet de dire si une opération contient une varaible **PARM_DECL** à droite, pour pouvoir spécifier une lecture d'adresse, et une autre fonction qui indique si l'opérande de gauche est un **MEM_REF** pour indiquer une écriture à l'adresse contenue dans l'opérande de gauche de l'opération.

Ces fonctions sont disponibles dans le fichier **Plugin/src/mihp_loop_analysis.cpp** ligne **111** et **157** respectivement.

4 La librairie d'analyse

La librarie d'analyse effectue l'analyse des accès mémoire du programme et va indiquer si la boucle est vectorisable ou non, elle prend en entrée les differentes adresses mémoire des variables manipulées durant chaque itération de la boucle, ainsi que le type de manipulation (lecture ou écriture) et compare dans un premier temps les adresses accédées entre une itération et ses suivantes ceci pour verifier si il y a des recouvrements entre les adresses mémoire (voir partie 4.2). Si il y a recouvrement, on teste le type de recouvrement en verifiant les types d'accés qui ont été effectués sur ses adresses (voir tableau XX).

Dans l'implémentation de notre librarie nous avons définie 3 classes : Mihp_Adress, Mihp_Iteration, Mihp_Loop pour recevoir les données envoyées grace au plugin, et pour permettre de faire l'analyse, voici ci-dessous un diagramme de nos classes (voir figure **??**).

Dans l'implémentation de notre librarie nous avons définie 3 classe : Mihp_Adress, Mihp_Iteration, Mihp_Loop pour recevoir les données envoyées grace au plugin, et pour permettre de faire l'analyse, voici ci-dessous un diagramme de nos classes :

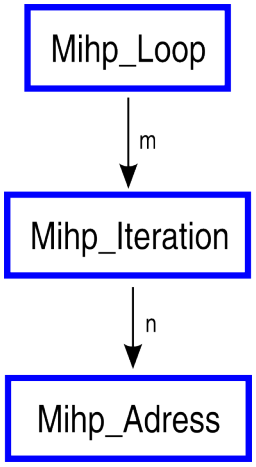

FIGURE 2 – classes de la librarie

5 Les tests

Pour tester le projet dans son ensemble et de manière efficace, nous avons choisi de permettre l'utilisation de la librairie d'analyse sans utiliser le plugin, ce qui nous a permis de débugger les deux parties de manière indépendantes. Nous avons donc réalisé des tests qui n'utilisent pas le plugin comme :

— TEST_WITHOUT_PLUGIN
— TEST_WITH_NO_PLUGIN
— TEST_WITHOUT_PLUGIN_VECTOR_SIZE

Ces tests nous on permis de vérifier trois cas différents :

— La vectorisation est complétement impossible
— La vectorisation est complétement possible
— La vectorisation est possible mais avec une taille de vecteur limité

En ce qui concerne le plugin, nous avons testé, les analyses de fonctions seules et multiples, le cas où il y a plusieurs boucles dans une fonction, le cas où il y a un nid de boucle. Seules les boucles contenues dans une condition contenu elle-même dans une autre boucle ne sont pas traité efficacement.

6 Analyse statique

Nous n'avons pas eu le temps d'implémenter cette dernière partie mais nous y avons quand même réfléchis. Une analyse statique ne peut être efficace dans tout les cas mais

nous l'avons vu comme un outil permettant d'alléger l'analyse dynamique. Car dans ce cas, seule les boucles n'ayant pas été résolues par l'analyse statique le seront par la dynamique.

Nous avons trouvé des conditions nécéssaires pour la vectorisation dans la documentation d'intel. Comme l'absence d'appel de fonction autre que les fonctions mathématiques usuelles (comme sin, cos, tan, etc), ou le fait que le nombre d'itération doit être prévisible.

7 Problèmes rencontrés

Les principaux problèmes rencontrés lors de l'impémentation du plugin ont été pour ajouter l'appel à la fonction **mihp_address**, car l'utilisation de la fonction **print_gimple_stmt** ne permet pas de vérifier dans tout les cas si le prototype de la fonction qui sera appelée par gimple correspond aux paramètre effectivement passés.

8 Conclusion

Notre biliothèque fournie une très bonne analyse dynamique, mais nous n'avons pas eut le temps de faire l'analyse statique avec le plugin. Les cas litigieux contenant des conditions contenant elles mêmes des boucles ne sont pas gérés, mais les nids de boucles simple et les boucles contenant des conditions simples sont traités.

Les sources du projet sont également disponibles sur le site https://github.com/salimus15/cpaproject.