Analyse des impacts énergétiques du Projet

GL22 G5

1. Consomation énergétique du projet

Dès le début du projet, notre équipe a entamé des discussions sur la réduction de la consommation énergétique de notre projet. Cela nous a également aidés à choisir notre extension OPTIM.

1.1 Motivation

La motivation principale derrière la réduction de l'impact énergétique du projet est la réduction de la production de carbone. En effet, le monde de l'informatique contribue également aux crises écologiques que nous rencontrons aujourd'hui dans le monde. Pour cela, prenons conscience de tous les risques climatiques qui sont liés à l'utilisation intensive de l'énergie et en les appliquant dans l'industrie de l'informatique, nous allons au moins aider à maintenir la stabilité de notre planète.

1.2 Méthodes d'évaluation de la consomation énergétique

Principalement, pour évaluer la consommation énergétique, on compte le nombre de cycles des instructions dans le fichier **.ass** généré à partir du fichier **.deca**. De plus, on a utilisé la commande shell **time** pour mesurer le temps d'exécution d'un fichier **.ass** en utilisant **ima**. Les fichiers de test **.deca** sont des fichiers qui contiennent des tests assez généraux (différentes opérations énergétiques, des boucles while, etc.) pour obtenir des résultats assez significatifs.

2. Stratégie mise en œuvre

Les stratégies utilisées sont principalement basées sur la réduction du code généré, l'optimisation des opérations d'affectation et d'initialisation et la diminution des branchements utilisés dans le code assembleur.

2.1 Les affectations et les initialisations de variables

le code ci dessous représente une inialisatoins de deux variable de type **float** on utilisant des operations aréthemetiques.

```
{
    float x = 140;
    float y = x + 7;
}
```

Un code Assambleur qui traduit ce code Deca:

```
LOAD #140, R2
STORE R2, 1(GB)
LOAD #7, R2
LOAD 1(GB), R3
ADD R3, R2
STORE R2, 2(GB)
```

Ce code utilise autant de registres que d'opérations arithmétiques dans le code. Par conséquent, un grand nombre d'opérations dans une initialisation ou affectation de variable nécessite un nombre énorme de registres.

Notre approche, que nous avons utilisée, réduit ce nombre de registres à un seul, quel que soit le nombre d'opérations.

Par example pour le code Deca suivant:

```
{
    float x = 30*4 + 20 + 7;
}
```

Le code assembleur généré, c'est

```
LOAD #30, R2
MUL #4, R2
ADD #20, R2
ADD #7, R2
STORE R2, 1(GB)
```

Ce code utilise un seul registre **R2** pour effectuer les 3 opérations arithmétiques dans l'initialisation de la variable x.

D'autres approches d'optimisation sont incluses dans l'extension du projet OPTIM. Plus de détails sont disponibles dans la prochaine section et dans la documentation de l'extension.

2. l'impact énergétique l'extension

Dans notre projet, nous avons choisi l'extension OPTIM qui a le plus d'impact énergétique parmi les extensions proposées pour ce projet. Cette extension permet de générer un code optimisé en termes de cycles utilisés dans les instructions, tout en gardant les mêmes fonctionnalités du compilateur. Voici quelque grand point des optimisations utilisées dans l'etension OPTIM:

2.1 Constant Folding

L'optimisation de Constant Folding est une technique de compilation qui consiste à calculer les expressions constantes à l'avance, au moment de la compilation, plutôt que de les calculer à l'exécution. Permettant de réduire le nombre d'opérations dans le programme. L'étude [1] a montré, à travers l'analyse de centaines de programmes Java, qu'en moyenne, le Constant Folding a permis de réduire la consommation d'énergie de 2,5 %.

2.2 Dead Code Elimination

La suppression de code mort (Dead Code Elimination, DCE) est une technique d'optimisation de code qui vise à supprimer les instructions ou les blocs de code qui n'ont aucun effet sur le comportement du programme. En éliminant ces instructions inutiles, le code devient plus petit, plus rapide et plus facile à maintenir. Cette méthode d'optimisation peut avoir un impact significatif sur la consommation d'énergie des programmes. En supprimant les instructions et les blocs de code qui ne seront jamais exécutés, les compilateurs peuvent éviter d'effectuer les opérations associées à ces instructions. Cela peut se traduire par une réduction de la consommation d'énergie du programme, en évitant de faire travailler les circuits du processeur pour rien.

2.3 Liveness Analysis

Cette opération permet de déterminer les variables vivantes avant chaque boucle while. Ces variables peuvent être stockées dans des registres, ce qui permet d'éviter les accès mémoire superflus dans la pile. En outre, les accès mémoire sont une source importante de consommation d'énergie. L'analyse de vivacité peut contribuer à réduire la consommation d'énergie des programmes en évitant les accès mémoire superflus.

2.4 Strength Reduction

"Strength Reduction" est une technique d'optimisation de code qui permet de réduire la consommation d'énergie des programmes. Elle consiste à remplacer des opérations coûteuses par des opérations équivalentes mais moins coûteuses. Dans notre compilateur, les multiplications et divisions par des puissances de 2 sont remplacées par des opérations de décalage.

2.5 Strength Reduction

Cette technique permet de réduiser le nombre de sauts dans les boucles "while" on les remplaçants par des "if + so while".

Conclusion

Dans ce projet, notre équipe a accordé une grande importance à l'optimisation de la compilation, notamment en réduisant la consommation énergétique de ce compilateur lors de son utilisation par les développeurs, ce qui contribuera à réduire l'impact environnemental des technologies informatique.

Documentation

[1]: Compiler-assisted energy reduction of java real-time programs by Manish Tewary, Zoran Salcic, Morteza Biglari-Abhari , Avinash Malik