PROJET GÉNIE LOGICIEL

2023/2024

Documentation sur les Impacts Énergétiques du Compilateur

Antoine Borget
Pierre Lin
Anass Haydar
Yassine Safraoui
Jihane Zemmoura

Groupe GL-47

Table des matières

1 Introduction

Dans le contexte de l'optimisation du code généré par un compilateur, notre approche vise à maximiser l'efficacité énergétique de l'exécution des applications développées par un grand nombre de programmeurs.

À travers une série d'optimisations détaillées, nous avons évalué notre performance en utilisant des scripts spécifiques, obtenant un score compétitif parmi les groupes des années précédentes.

Les optimisations du code généré pour la machine IMA comprennent de nombreuses stratégies que nous allons présenter dans une partie dédiée.

Bien que des efforts aient été consacrés aux optimisations pour l'extension GameBoy, les possibilités étaient limitées en raison de contraintes temporelles et techniques.

En parallèle, nous allons souligner l'importance d'optimiser le code qui génère le compilateur lui-même, en mettant en avant des choix judicieux d'algorithmes et de structures de données pour maximiser l'efficacité, même dans des aspects moins évidents du processus de compilation.

2 Optimisations sur le code généré

Comme un compilateur est fait pour être utilisé par des milliers (voire des millions) de programmeurs, qui eux-même vont utiliser ce compilateur pour développer des applications utilisées par encore plus de personnes, il est primordial d'optimiser le code généré par le compilateur afin d'utiliser le moins d'énergie possible pour l'exécution de chaque applications. Nous avons donc décidé de faire plusieurs optimisations sur le code généré.

2.1 Score

Nous avons évalué l'efficacité du code généré grâce aux scripts donnés : *syracuse42*, *ln2*, et *ln2_fct*. Pour avoir le temps d'exécution de ces programmes, nous avons utilisé l'option *-s* de la machine IMA. Les résultats pour chacun des programmes cités précédemment sont respectivement : 686, 8468, et 11223 unités. Grâce à ces temps, nous avons pû nous comparer aux groupes des années précédentes sur ce site. Pour calculer notre score, nous avons utilisé la formule donnée sur le site :

$$10 \times S + L_0 + L_1 = 10 \times 686 + 8468 + 11223 = \mathbf{26551}.$$

Pour chacune des années sur le site, nous arrivons toujours $3^{\text{ème}}$ parmi tous les groupes. Ce score est très satisfaisant, considérant le fait que notre extension est ACONIT (qui n'a pas de rapport avec l'optimisation du code généré).

2.2 Optimisations pour la machine IMA

Dans cette partie, nous allons présenter dans une liste quelques optimisations que nous avons faites sur le code généré. Pour chaque point de la liste, il y aura une petite description de l'optimisation, puis, un exemple sous la forme d'une tableau où :

- La $1^{\text{ère}}$ colonne représente une instruction d'un programme Deca.
- La $2^{\text{ème}}$ colonne représente notre code généré optimisé.
- La $3^{\text{ème}}$ colonne représente ce qu'était notre code généré avant optimisation.

Enfin, nous donnerons le nombre de cycles gagnés dans l'exemple grâce à l'optimisation faite.

On supposera que la variable x est stocké dans le registre R2 ou dans la case 2(GB) selon les exemples. Voici la liste de nos optimisations :

1. Suppression des LOAD de constantes inutiles dans les opérations arithmétiques.

$x * 6$	Mul #6, R2	Load #6, R3 Mul R3, R2

   Résultat : Gain de 2 cycles.

2. Simplification des opérations arithmétiques entre constantes.

$(12 * 6) + x$	Add #72, R2	Load #12, R3
		Mul #6, R3
		Add R3, R2

Résultat : Le gain dépend de l'opération entre les deux constantes.
Ici, on gagne $20 + 2 = 22$ cycles.

Note : Si il y a une division par 0, ou un débordement pour l'opération entre deux *float* constants, on fait directement un branchement vers le label de l'erreur concernée via l'instruction BRA.

3. Utilisation des shifts SHL et SHR pour la multiplication/division entière par une constante du type 2^k où $k < 10$.

$x * 8$	Shl R2 Shl R2 Shl R2	Mul #8, R2

Résultat : Ici, nous avons un gain de 14 cycles.

Note : Si $k \geq 10$, alors nous utilisation MUL/QUO, car sinon, pour avoir le résultat, il faudrait faire plus de 10 SHL/SHR, ce qui devient moins efficace. Dans le cas $k = 10$, nous avons choisi MUL/QUO car le code généré est plus court.

4. Suppresion des CMP inutiles pour une comparaison à 0 après une opération. En effet, comme les LOAD/STORE, et les instructions d'opération arithmétique en IMA placent déjà les flags de comparaison pour une comparaison à 0, certains CMP sont inutiles.

if $(x > 0)$ { } else { }	Store 2(GB), R2 Ble elseLabel	Store 2(GB), R2 CMP 0, R2 Ble elseLabel

Résultat : Gain de 2 cycles.

5. Malgré le fait qu'il n'y ai pas d'instruction AND en IMA qui aurait permit d'optimiser le modulo par une puissance de 2 facilement, nous avons trouvé important de quand même gérer le cas du modulo par 2. En effet, le modulo par 2 est une opération très utilisée dans les programmes pour tester si un nombre est pair ou non.
Pour cela nous avons utilisé le fait que $x \,\%\, 2 = x - (x/2) \times 2$, où la division est entière.
- Par exemple, dans le cas pair :
$42 \,\%\, 2 = 42 - (42/2) \times 2 = 42 - 21 \times 2 = 42 - 42 = 0.$
- De même, dans le cas impair :
$41 \,\%\, 2 = 41 - (41/2) \times 2 = 41 - 20 \times 2 = 41 - 40 = 1.$
Les résultats sont bons dans les deux cas. Comme ce sont des opérations de multiplications/divisions par 2, on peut utiliser les shifts, qui sont très efficaces.

$x \% 2$	Load R2, R3 Shr R3 Shl R3 Sub R3, R2	Rem #2, R3

Résultat : Gain de 32 cycles.

Note : Nous aurions pu faire quelque chose du même type pour le modulo par 4, 8, etc... Cependant, nous n'avons pas estimé cela rentable par rapport au niveau d'utilisation du modulo par ces entiers, et notre temps restant pour terminer le projet.

6. Pour l'évaluation des expressions booléennes, nous avons implémenté l'évaluation paresseuse de ces expressions. Cette implémentation a été faite en accord avec ce qui est décrit à partir de la page 220 du polycopié.
   Par exemple, dans l'expression $e_1 \mid\mid e_2$, si e_1 est vrai, on n'évalue pas e_2, et on branche directement au label destiné au cas vrai.

2.3 Optimisations pour l'extension GameBoy

Pour le code généré sur la GameBoy, nous n'avons pas pu faire beaucoup d'optimisations, car il était déjà très difficile d'avoir quelque chose de fonctionnel. En plus de cela, le nombre d'instructions assembleurs pouvant nous aider à l'optimisation du code généré était très réduit. Néanmoins, les évaluations paresseuses d'expressions booléennes restent présentes.

3 Optimisations du code générant le compilateur

En dehors du fait d'avoir optimisé le code généré par le compilateur. Nous avons aussi trouvé intéressant d'optimiser le code que nous écrivons pour générer le compilateur. Bien que l'échelle est beaucoup moins grande car il n'y a que nous qui utilisons le code pour générer le compilateur, il n'en reste pas moins que l'impact énergétique est là.

3.1 Optimisations du code de l'étape C

Cette partie va se concentrer sur les optimisations faite pour l'étape C.

Pour le choix des algorithmes et des structures de données, nous avons bien fait attention à les choisir le plus efficacement possible, et donc que toutes les opérations soient en $\mathcal{O}(1)$. Tous ces choix sont décrits dans la partie concernant l'étape C du document concernant la conception du compilateur. Mais, par exemple, au début du projet, pour stocker les registres, nous avions eu recourt à un tableau de booléen qui disait pour chaque cases, si oui ou non, le registre était libre. L'accès à un registre libre était alors en $\mathcal{O}(n)$. Après réflexion, nous avons remplacé cela par une liste chaînée représentant une pile LIFO (Last In First Out) de registres disponibles. Si on veut avoir un registre libre, on retire la tête en $\mathcal{O}(1)$, et, si on veut libérer un registre, on le ré-ajoute en tête en $\mathcal{O}(1)$.

Un autre exemple est lorsque l'on voulait trier les méthodes d'une classe en fonction de leur index pour générer la table des méthodes. Au début, nous voulions le faire avec un tri en $\mathcal{O}(n\log(n))$ comme le tri fusion, mais nous nous sommes rendu compte que cela allait être moins efficace qu'un tri en en $\mathcal{O}(n^2)$. En effet, dans la plupart du temps, le nombre de méthode d'une classe est très réduit, dans ces cas là, les tris en $\mathcal{O}(n^2)$ s'avèrent plus efficaces de par leur simplicité. C'est pourquoi nous avons opté pour un tri sélection malgré sa complexité de $\mathcal{O}(n^2)$.