Projet GL 2021:
Bilan de l'impact énergétique

Équipe n°31 : Antoine Briançon, Axel Glorvigen, Thibault Launay, Maxime Martin et Pierre Pocreau

Sommaire:

I. Introduction

Les enjeux climatiques sont aujourd'hui une problématique globale, présents dans tous les secteurs d'activités. Le domaine de l'informatique, aussi bien au niveau matériel que logiciel, est responsable d'une part importante de la pollution mondiale, aussi bien en consommation d'énergie qu'en production de déchets. Il est donc essentiel que ces enjeux soient enseignés aux futurs ingénieurs du numériques, afin de trouver des solutions efficientes et durables face à l'essor du numérique dans le monde entier. Dans le cadre du Projet GL 2021 à Grenoble INP - Ensimag, nous avons mis en place certaines solutions techniques pour limiter l'impact énergétique de notre travail et de notre produit. Ces solutions sont variées, allant des optimisations du code jusqu'à des méthodes de travail moins polluantes, et peuvent sembler dérisoires à l'échelle d'un individu. Cependant, elles peuvent avoir des répercussions intéressantes dans le cadre d'utilisations massives d'un produit logiciel. L'exemple le plus probant est celui du Bitcoin, dont les serveurs consomment à l'année plus d'électricité que la Belgique, mais dont l'impact serait bien pire si des techniques d'optimisation logicielle n'étaient pas utilisées. Nous présentons ainsi dans ce document une analyse de l'impact énergétique de notre compilateur Decac, ainsi que certaines solutions mises en place pour limiter son impact énergétique et environnemental.

II. Efficience du produit

La consommation d'énergie de notre compilateur peut être abordée selon deux aspects : la consommation nécessaire au fonctionnement du compilateur pour traduire un programme en langage Deca vers le langage assembleur, et l'efficience du code assembleur produit par le compilateur, ce code assembleur étant ensuite exécuté par une machine dont on aimerait limiter l'impact énergétique. Nous avons principalement travaillé sur le deuxième aspect de cette consommation d'énergie, en essayant de limiter le nombre de cycles effectués par le code assembleur et ainsi rendre les codes assembleur moins gourmands en énergie. Cet aspect est d'autant plus important dans le cadre d'une utilisation importante du compilateur par de nombreux clients, qui cherchent à obtenir un code exact et efficace à la sortie du compilateur.

Certaines optimisations peuvent se faire directement, lors de la première passe sur le programme, alors que d'autres nécessitent une deuxième passe pour optimiser le code assembleur déjà produit. Nous avons ainsi implémenté deux méthodes d'optimisation parmi les différents motifs de code assembleurs qui peuvent être optimisés, que l'on présente dans la suite de cette partie.

A. Store/Load

Lorsque l'on évalue une expression, celle-ci est calculée dans le registre courant. Quand on effectue une déclaration ou une assignation pour une variable, on effectue un "STORE", pour charger l'adresse mémoire de la variable avec la valeur que l'on vient de calculer. Si on l'utilise dans l'instruction suivante, le compilateur re-charger la variable dans un registre, parfois le même registre que celui dans laquelle l'expression a été calculée. Le schéma est le suivant:

STORE *Rx, y(LB)*
LOAD *y(LB), Rx*

La valeur étant déjà dans le registre *Rx*, nous pouvons supprimer l'instruction "LOAD" et ainsi gagner une instruction coûtant 2 cycles internes. Notre optimiseur est, à ce niveau, rudimentaire, et parcourt le programme final pour y trouver le motif précédent, c'est-à-dire sur deux lignes consécutives. On pourrait améliorer cet optimiseur pour supprimer l'instruction "LOAD" seulement si la valeur contenue dans Rx n'a pas été écrasée.

Pour mesurer l'impact de cette optimisation, nous avons lancé le programme ln2.deca pour le compilateur avec et sans l'optimiseur, et nous nous sommes rendu compte que nous avions économisé 288 cycles internes, soit environ 2% du nombre de cycles total, lorsque que le compilateur était associé à cette optimisation. Le résultat est donc satisfaisant et montre un véritable potentiel des optimisations au sein du compilateur Decac.

B. Permutation d'opérandes / Opérations commutatives

Lorsqu'une opération arithmétique binaire est codée, on évalue son expression de gauche dans le registre courant n, puis son expression de droite dans le registre n+1 (si ce registre est accessible). Le résultat de l'opération est stockée dans le registre n. Dans le cas où l'opérande de gauche est une variable, nous devons la charger dans un registre avant de réaliser l'opération, alors que nous pouvons réaliser une opération en utilisant uniquement une adresse de variable. Afin de mettre en place cette optimisation, dans l'idée de réduire le nombre d'instructions "LOAD", nous permutons les opérandes des opérations commutatives (addition et multiplication) lorsque l'opérande de gauche est un littéral ou une variable. On illustre ceci avec l'exemple de code assembleur suivant :

Optimisé

```
19|  LOAD #0x1.0p0, R2
20|  DIV #0x1.3bp12, R2
21|  MUL 5(GB), R2
22|  LOAD #0x1.0p0, R3
23|  DIV #0x1.68p9, R3
24|  ADD R3, R2
25|  MUL 5(GB), R2
26|  LOAD #0x1.0p0, R3
27|  DIV #0x1.ep6, R3
28|  ADD R3, R2
29|  MUL 5(GB), R2
30|  LOAD #0x1.0p0, R3
```

Non Optimisé

```
16|  ADD 2(GB), R2
17|  DIV #0x1.0p1, R2
18|  STORE R2, 5(GB)
19|  LOAD 5(GB), R2
20|  LOAD 5(GB), R3
21|  LOAD 5(GB), R4
22|  LOAD 5(GB), R5
23|  LOAD 5(GB), R6
24|  LOAD 5(GB), R7
25|  LOAD 5(GB), R8
26|  LOAD #0x1.0p0, R9
27|  DIV #0x1.3bp12, R9
28|  MUL R9, R8
```

Dans le code de droite, non optimisé, on répète plusieurs fois l'instruction "LOAD" pour charger la valeur stockée dans la pile globale, alors que dans le code de gauche, optimisé, on effectue directement les opérations en utilisant l'adresse de la pile.

Cette optimisation est très efficace car elle permet, sur l'exemple ln2.deca, d'économiser 330 cycles internes, soit environ 2.2% du nombre de cycles obtenus sans optimisation.

III. Efficience du procédé de fabrication

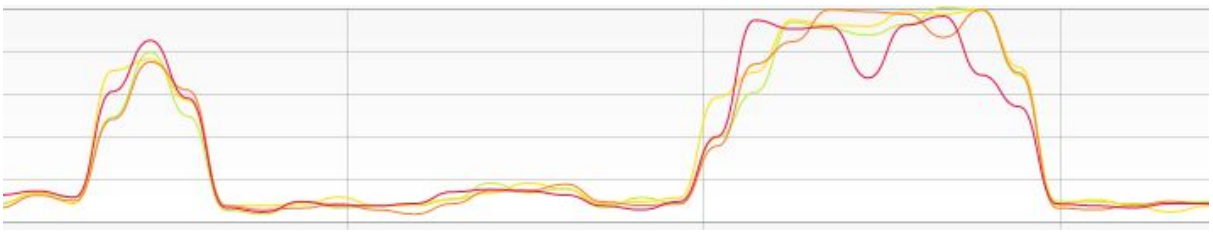
A. Optimisation des tests et compilation

La deuxième partie de l'optimisation porte sur la phase de test du programme. La consommation énergétique est importante durant les phases de test et de compilation, ce que l'on peut optimiser de plusieurs façons.

Pour la compilation du projet, nous utilisons l'outil "Run" de l'IDE Intellij car celui-ci est géré de manière intelligente et repère les fichiers modifiés par rapport à la dernière compilation, ce qui permet de seulement recompiler les fichiers nécessaires. L'outil Maven, au contraire, compile tous les fichiers à chaque lancement, avec la commande "mvn clean compile", ce qui n'est pas optimal d'un point de vue énergétique.

Nous proposons ainsi une illustration de cette différence de compilation avec le graphe ci-dessous. À gauche, le pic correspondant à une compilation avec Intellij, et à droite, avec l'outil Maven.

(Echelle: horizontale 10 secs, Verticale 20% d'utilisation du CPU)



On voit ainsi que les temps de compilation sont réduits, et que l'IDE Intellij permet une compilation plus rapide tout en consommant moins de ressources au niveau du CPU. Les outils de développement sont donc importants à prendre en compte pour minimiser l'impact énergétique d'un projet car ils permettent des économies non négligeables.

Pour le lancement des tests, nous avons des scripts exécutant tous les tests d'une partie (Hello World, Sans-Objet ou Complet). Ceci permet de ne pas relancer des tests des parties précédentes lors du développement d'une nouvelle partie, ce qui correspond à des économies de temps et d'énergie. En cas d'erreur, on utilise les tests manuels afin d'analyser sa provenance, dans le but de la corriger et de valider les tests. Ce procédé est bien plus efficace que de relancer tous les tests à chaque fois que l'on modifie un morceau de code.

B. Economies générales

Nous utilisons l'outil Git, pour le partage et la mise en commun de nos différents codes. Afin d'utiliser Git de manière efficace, nous avons créé plusieurs branches (deb-B, dev-C, dev-Trigo, etc…), ce qui nous a permis de travailler sur certaines parties spécifiques du projet, sans avoir à récupérer l'avancement des autres branches qui n'étaient pas essentielles à la nôtre. Cette technique, facilitant l'organisation et permettant un impact environnemental moindre en évitant la lourde commande "git pull", est très simple à mettre en place et constitue un atout très intéressant de Git.

Nous travaillons également tous directement sur les OS de nos machines, ce qui est un gain en performances mais évite aussi d'utiliser une VM, lourde et plus gourmande en énergie.

IV. Conclusion

Nous avons pris plusieurs mesures pour limiter notre impact énergétique sur toute la durée du projet, cependant, la plus grande partie de l'impact du compilateur se fait pendant son utilisation par potentiellement plusieurs milliers d'utilisateurs. Ainsi, l'optimisation du code assembleur produit par le compilateur est le point clé qui limite sa consommation énergétique. Il existe de nombreuses méthodes d'optimisation qui peuvent être mises en oeuvre, nous en avons implémenté deux, aller plus loin dans cette optimisation nécessiterai plus de temps.