École nationale de l'informatique et des mathématiques appliquées de Grenoble

Projet Génie Logiciel

Manuel Analyse de l'impact énergétique
Année universitaire 2020-2021

Groupe 8 Equipe 40
Étudiants 2A

Table des matières

1 Moyens mis en oeuvre pour évaluer la consommation énergétique

1.1 Mesure des performances

Un premier outil utile à la mesure de notre consommation énergétique est le calcul de nos performances à mettre en comparaison avec les résultats des autres équipes (résultat disponible dans le palmarès) .

1.2 ima -s nombre d'instruction et temps d'execution

Afin de comprendre la consommation énergétique de notre projet nous avons décidé d'utiliser l'option *-s* de *ima* afin de regarder combien de cycles produisent nos tests les plus imposants et les plus complets*.

*tests en question disponibles dans les fichiers suivants :

- src/test/deca/codegen/valid/objet/exDiapos.deca
- src/test/deca/codegen/valid/objet/MutlipleHeritage.deca
- src/test/deca/biblio/morpion.deca
- src/test/deca/codegen/valid/extension/mat.deca
- src/test/deca/codegen/valid/biblio/Testvlp.deca

1.3 Mesure de temps des différentes étapes

Pour pouvoir analyser notre consommation énergétique, on essaie de trouver la partie la plus consommatrice de notre projet. On suppose que la partie la plus consommatrice est la partie C de génération de code, mais pour le démontrer nous allons utiliser la commande *time* pour comparer les différents temps du point de vue de l'utilisateur et du point de vue du système sur les fichiers tests précédents.

2 Coût énergétique

2.1 Mesure des performances

Syracuse42	ln2	ln2_fct
1404	18404	22839

2.2 ima -s

2.2.1 ima -s nombre d'instructions

Nom du test	Résultat de ima -s
exDiapos.deca	178
MutlipleHeritage.deca	333
morpion.deca	1072
mat.deca	643
Testvlp.deca	5129

2.2.2 ima -s temps d'execution

Nom du test	Résultat de ima -s
exDiapos.deca	848
MutlipleHeritage.deca	2822
morpion.deca	86301 (en moyenne)
mat.deca	214063
Testvlp.deca	2071773

2.2.3 Ratio instructions-temps

Nom du test	Ratio(temps/instruction)
exDiapos.deca	4,76
MutlipleHeritage.deca	8,47
morpion.deca	80,5
mat.deca	332,9
Testvlp.deca	403,93

L'extension semble très "gourmande" en énergie (sans doute dû au fait qu'on ne considère que les matrices de flottant, plus consommatrice que des matrices d'entiers, point développé par la suite).

2.3 Mesure de temps des différentes étapes

2.3.1 Time, le temps utilisateur

Nom du test	lexer	parser	contexte	gencode
exDiapos.deca	0,268	0,348	0,365	0,340
MutlipleHeritage.deca	0,283	0,346	0,421	0,363
morpion.deca	0,317	0,411	0,446	0,528
mat.deca	0,327	0,439	0,434	0,446
Testvlp.deca	0,533	0,745	1,131	0,844

2.3.2 Time, le temps système

Nom du test	lexer	parser	contexte	gencode
exDiapos.deca	0,045	0,032	0,026	0,048
MutlipleHeritage.deca	0,68	0,045	0,042	0,037
morpion.deca	0,038	0,032	0,059	0,077
mat.deca	0,018	0,054	0,041	0,040
Testvlp.deca	0,033	0,097	0,113	0,085

2.3.3 Conclusion de time

Avec ces mesures, on peut remarquer à nouveau que la partie extension est la plus consommatrice d'énergie. Cependant, avec ces mesures on ne remarque pas d'énormes différences entre les différentes parties mais la partie génération de code reste quand même la partie la plus

complexe et donc celle qui serait sans doute la plus intéressante à optimiser pour réduire notre impact énergétique.

3 Diminuer l'impact énergétique

3.1 Concernant les test

3.1.1 Les fichiers modèles

De nombreuses parties de notre projet sont tout d'abord testées lors de l'exécution d'un test puis le résultat de cette exécution est comparé avec un modèle mis en place au préalable représentant le résultat que nous attendions. Bien que cette manière d'opérer permette de s'assurer que nos programmes font le travail attendu, la présence de nombreux fichiers modèles est requise (nommé en général *deca.ok, ass.ok* ou *deca.lex.ok*).

De plus, certains tests (particulièrement ceux concernant l'exécution et la compilation avec *ima*) aboutissent à un résultat vide, car nous ne renvoyons rien. Cependant, nous avons construit nos scripts de telle sorte que, dans les parties concernées, tout fichier *.deca* que nous exécutons doit avoir un fichier du même nom finissant par *ass.ok*, ou *deca.ok*, pour être comparé. Nous avons donc la présence de fichiers modèles vides mais nécessaire à la réalisation de nos scripts.

Dans le temps imparti nous n'avons pas réussi à trouver une alternative à ce problème de nombreux fichiers modèles. C'est un des points que nous pourrions améliorer car moins de fichiers à stocker implique moins d'énergie consommée. Une première solution serait de "marquer" les fichiers retournant un résultat vide pour tous les comparer avec un même modèle (avec par exemple une phrase reconnaissable en première ligne du fichier). Cela permettrait de ne pas stocker des fichiers vides inutilement.

Une autre solution dans un second temps serait de trouver une manière de s'assurer que nos programmes font le travail demandé sans devoir stocker un fichier modèle entier. Cette solution, bien qu'évoquée au sein du groupe, n'a pas encore abouti à une réflexion complète.

3.1.2 Les fichiers tests

Au cours du projet, tous les membres du groupe ont participé à la mise en place des tests soit en les écrivant directement soit en donnant des idées de tests plus complexes auxquels ils auraient pensé. Au début du projet, il était simple de comparer les idées aux tests déjà mis en place, mais plus les parties ont avancé et plus il a été difficile de dire exactement quelles idées avaient déjà été exploitées.

De plus, certains tests résumaient des tests antérieurs mais apportaient tout de même une nouveauté. Ils étaient donc aussi mis en place.

Enfin, certaines parties et leurs tests étant faites par des personnes différentes, chacun faisait des tests qui lui semblait nécessaire pour avancer dans sa partie et mettre en place son code. Lors de la réunion des différentes parties, il a donc été remarqué que certaines personnes avaient pensé à des tests différents (tests qui ont donc été appliqués aux autres parties) mais aussi des tests qui se répétaient.

Il a été décidé de garder pratiquement tous les tests qui ont été écrits lors du projet. Tout d'abord car cela permettait de s'assurer qu'une grande majorité du code était testée. Cela serait d'autant plus appréciable dans le cas où une personne extérieure voudrait s'insérer dans le projet et donc comprendre rapidement quelle partie était testée et comment. De plus, dans le contexte du projet scolaire, nous nous sommes dit qu'il était préférable de pouvoir voir et comprendre notre cheminement de penser à travers nos tests.

Cependant, nous sommes rapidement arrivés à un très grand nombre de tests dans notre projet ce qui entraîne de nombreux tests répétitifs et ainsi de nombreux fichiers tests stockés et testés à chaque *mvn test* (et donc une consommation énergétique grandissante).

Une solution à ce problème serait dans un premier temps de vérifier nos tests et de supprimer les tests répétitifs dans chaque partie.

Dans un second temps, nous nous sommes rendus compte que l'exécution et la compilation avec *decac* et *ima* résumait l'entièreté du projet. Il serait donc intéressant de faire un dossier de tests résumant nos tests actuels, de créer un script pour cela et de n'exécuter que ce script lors de nos *mvn test*. Ainsi toutes les parties du projet seraient testées avec moins de fichiers tests stockés.

Cependant, nous lançons *mvn test* avant chaque push sur git (pour s'assurer que le travail précédent marche toujours), notre consommation énergétique lors de cette exécution en resterait inchangée et nous n'avons pas encore trouvé une solution à ce problème de consommation fréquente d'énergie.

3.2 Concernant la génération de code

Au niveau de la place en mémoire prise par un fichier .ass et de la lourdeur d'écriture, les branchements lors d'un *if then else* sont gérés de telle manière que pour chaque *if* on introduit un label de fin de condition. Ainsi, pour un code comme ceci :

```
1 if (a) {
2 } else if (b) {
3 } else if (c) {
4 } else {
5 }
```

Il y aura 3 labels de fin pour les conditions *a, b, c* afin de brancher la fin des instructions dans un des *if* vers la fin de cette instruction. Le plus optimal aurait été de réussir à n'avoir qu'un seul label pour l'ensemble des *if else if else* mais cela n'a pas été fait au vu du caractère imbriqué en *if then else* d'une instruction *if else if else*. Le code ci-dessus se traduit par exemple en :

```
if (a) {
} else {
    if (b) {
    } else {
        if (c) {
        } else {
        }
    }
}
```

et chaque instruction dans les *else* est traitée indépendamment de si elle est un nouveau *if else* ou n'importe quoi d'autre.

```
; ------------------------------------------------
;   Construction des tables des methodes
; ------------------------------------------------
; Construction de la table des methodes de Object
  LOAD #null, R0
  TSTO #1
  BOV pile_pleine
  ADDSP #1
  STORE R0, 1(GB)
  LOAD code.Object.equals, R0
  TSTO #1
  BOV pile_pleine
  ADDSP #1
  STORE R0, 2(GB)
; Construction de la table des methodes de A
  LEA 1(GB), R0
  TSTO #1
  BOV pile_pleine
  ADDSP #1
  STORE R0, 3(GB)
  LOAD code.Object.equals, R0
  TSTO #1
  BOV pile_pleine
  ADDSP #1
  STORE R0, 4(GB)
  LOAD code.A.getX, R0
  TSTO #1
  BOV pile_pleine
  ADDSP #1
  STORE R0, 5(GB)
  LOAD code.A.setX, R0
  TSTO #1
```

```
33  BOV pile_pleine
34  ADDSP #1
35  STORE R0, 6(GB)
```

Abordons à présent les tests de débordement de pile. Ces derniers sont nécessaires pour chaque instruction ayant un effet sur la pile (*PUSH, STORE, BSR..*). Initialement, nous faisions un test de débordement de pile (*TSTO*) avant chacune de ces instructions. Rapidement, nous avons évidemment souhaité factoriser ces instructions, dans un but d'optimisation du code généré, limitant aussi l'impact énergétique. Ainsi, nous avons mis en place des compteurs et utiliser les possibilités d'ajout de commande en tête de programme (*addFirst*) et de concaténation de programme (*append*) pour parvenir à nos fins. Cela a permis de passer, par exemple, pour la déclaration de classes et méthodes (table des méthodes), de codes comme celui ci-dessus, à des codes comme celui ci-dessous :

```
1   TSTO #11
2   BOV pile_pleine
3   ADDSP #7
4  ; ------------------------------------------------
5  ;    Construction des tables des methodes
6  ; ------------------------------------------------
7  ; Construction de la table des methodes de Object
8   LOAD #null, R0
9   STORE R0, 1(GB)
10  LOAD code.Object.equals, R0
11  STORE R0, 2(GB)
12 ; Construction de la table des methodes de A
13  LEA 1(GB), R0
14  STORE R0, 3(GB)
15  LOAD code.Object.equals, R0
16  STORE R0, 4(GB)
17  LOAD code.A.getX, R0
18  STORE R0, 5(GB)
19  LOAD code.A.setX, R0
20  STORE R0, 6(GB)
```

On voit dans cet exemple très simple un gain conséquent de 15 lignes d'instructions (il n'y avait qu'une seule classe et deux méthodes déclarées). Cette factorisation n'a cependant pas encore été poussée à son potentiel maximum : il demeure toujours des instructions, dans le bloc main par exemple, où l'on fait des tests *TSTO* "unitaires".

Dans l'optique de produire le compilateur fonctionnel dans les temps, certaines parties de conception ne sont pas optimales, mais nous les avons listées et ces dernières peuvent être reprises rapidement, d'autant plus que nous avons une meilleure vision globale du compilateur fini. Parmi celles-ci, nous pouvons notamment relever notre façon de générer les corps de méthodes. En effet, avant chaque appel de méthode, on doit préserver les registres qui vont être

utilisés (autres que R0 et R1). Toujours dans l'optique de concevoir rapidement un compilateur fonctionnel, nous avions omis dans un premier temps les possibilités décrites précédemment d'ajouter des instructions en tête de programme. C'est pourquoi nous avons fait un mauvais choix de conception : comme il est impossible de savoir les registres utilisés avant d'avoir écrit le corps de la méthode, nous pensions que nous étions obligés de faire deux générations du corps d'une méthode, une première pour voir les registres utilisés, une seconde effective. Cela était bien sur une erreur que nous pouvons à présent corriger (entre la soutenance et le rendu).

Enfin, on peut aussi évoquer des détails qui peuvent optimiser le nombre d'instructions assembleurs. Par exemple, à l'issue d'un appel de méthode, on fait systématiquement une instruction de la forme *LOAD(R0, Rn)*, ce qui est parfois absurde si n vaut 0 (ce qui peut être le cas d'une méthode en appelant une autre..). Il faudrait donc vérifier avant chaque LOAD que les deux registres en jeu sont différents, pour éviter ces instructions inutiles.

4 Prise en compte de l'impact énergétique dans l'extension

Notre choix de conception de tableau, décrit dans le document extension, permet d'allouer moins d'adresses que la première méthode que nous avions envisagé (pointeur en chaîne, cf document extension 3.4), ce fût aussi un argument renforçant ce choix (même si allouer un seul bloc pour toutes les valeurs peut devenir problématique car cela nécessite des adresses consécutives sur le tas). Cependant, un gros défaut de notre conception (en l'état, cela peut être rapidement résolu) est l'impossibilité de libérer les tableaux alloués sur la tas. Ainsi, de la même façon que pour le langage C il faudrait une commande *free*, car en l'état un programme Deca utilisant des tableaux peut vite remplir le tas, ce qui n'est pas optimal.

Enfin, avec les quelques jours de recul post-rendu pour préparer notre présentation de soutenance, nous avons réalisé que nous avions commis une erreur dans l'un de nos choix pour les bibliothèques. En effet, initialement nous voulions proposer deux classes de matrice : *MatFloat* et *MatInt*. Comme leurs noms le laissent entendre, l'une gère les matrices de float et l'autre les int. Mais en fait, cela nous paraissait très redondant, d'autant plus qu'une MatFloat peut être initialisée avec un tableau d'entiers. Nous avions donc choisi de supprimer TabInt. Mais cela était une erreur du point de vu énergétique. En effet, un utilisateur qui sait qu'il ne va gérer que des entiers, est dès lors obligé d'utiliser des flottants, qui ont beaucoup plus de défauts : plus de place en mémoire, et beaucoup de vérifications (tests de débordements). Cela est probablement l'une des explications des résultats obtenus en partie 2.2.3 Ratio instructions-temps.