Projet Génie Logiciel

Analyse des Impacts énergétiques

Aurélien VILMINOT
Damien CLAUZON
Guilherme KLEIN
Léon ROUSSEL
Pierre ARVY

26 janvier 2022

Résumé

Ce document est le fruit de notre réflexion quant à l'efficience de notre projet et de son produit, le compilateur `decac`. L'analyse énergétique se divise en deux parties : l'analyse de l'efficience du code produit et l'analyse de l'efficience du procédé de fabrication.

Table des matières

1 Préambule **2**

2 Code Produit **2**

2.1 Optimisations effectuées 2

2.2 Optimisations envisageables 2

2.2.1 Code mort 2

2.2.2 Conditions booléennes 3

2.2.3 Expressions arithmétiques 4

2.3 Mesures 4

3 Procédé de fabrication **5**

1 Préambule

Ces quelques pages proposent une réflexion sur les impacts énergétiques de notre projet. Les deux critères d'évaluation considérés seront l'efficience du code produit et du procédé de fabrication. Sachant que lors de l'exécution d'un programme informatique, le CPU est à l'origine de la majorité de l'énergie consommée par l'ordinateur (89% en moyenne, comme sur machine virtuelle)[1], l'évaluation de la performance du compilateur sera faite en nombre de cycles internes. Cette évaluation n'est pas limitante, sachant que la consommation énergétique est liée aux opérations internes effectuées et non à l'utilisation de la mémoire[2]. Le facteur important est en effet la vitesse d'exécution. Il suffit de revenir à la formule du calcul de l'énergie pour s'en convaincre :

$$E = T \times P$$

Malgré cela, la majorité des tests ont été lancés sur des ordinateurs portables, consommant trois fois moins d'énergie qu'un ordinateur de bureau en moyenne[3].

2 Code Produit

2.1 Optimisations effectuées

Pendant la génération du code, plusieurs registres sont utilisables et seuls les deux premiers sont réservés à des utilisations particulières. Notre objectif est d'utiliser le moins de registres possible cela permet principalement de résoudre deux problèmes :

1. Réduire le nombre de transferts de données à effectuer pendant des calculs.
En effet, le fait de générer le résultat des calculs directement dans le registre qui sera utilisé allège des instructions intermédiaires nécessaires pour transférer le contenu d'un registre dans un autre. Par exemple, dans le cas de l'instruction Deca `print`, les expressions à afficher sont directement calculées dans le registre R_1 qui est utilisé pour l'affichage.

2. Réduire le nombre de sauvegardes de registre à effectuer.
Quand le code relatif à une expression est généré, elle est enregistrée dans le plus petit registre disponible (en commençant par R_2), ce qui permet d'atteindre rarement le plus grand registre et donc d'éviter des sauvegardes de registre pour des calculs intermédiaires. Cela a aussi un impact dans l'initialisation des classes et dans le code des méthodes, car il est nécessaire de sauvegarder tous les registres utilisés au cours de ces deux appels. Cela réduit ainsi l'utilisation de la pile et donc de la mémoire.

Nous avons également pris soin de minimiser le nombre de tests de débordement de la pile et le nombre d'ajouts au pointeur de la pile en mémorisant au cours de la génération du code la taille de pile minimale nécessaire pour stocker toutes les variables à enregistrer dans la pile, ainsi que le nombre de variables globales et locales. Ainsi, les instructions ADDSP et TSTO ne sont utilisées qu'au début du programme et au début de l'initialisation des classes et du code des méthodes au lieu de les répéter avant chaque sauvegarde dans la pile.

2.2 Optimisations envisageables

Outre les optimisations réalisées sur la génération de code, d'autres sont possibles et réalisables sur le projet de Génie Logiciel.

2.2.1 Code mort

La première de ces optimisations concerne la présence de code mort à l'intérieur d'une condition booléenne. Un exemple contenant du code mort peut être :

```
{
   if (false) {
      println("foo");
   }
}
```

En effet, quel que soit le contexte dans lequel ce code est exécuté, l'instruction `println("foo")` ne sera jamais exécutée par la machine abstraite. Il n'est, par conséquent, pas nécessaire de le générer lors de la génération de code. Actuellement, deux passes sont effectuées lors de l'étape de génération de code. La première passe consiste à parcourir les classes du programmes afin de construire la table des méthodes. La seconde passe concerne quant à elle le codage complet des classes et du programme principal. Pour réaliser l'optimisation, il serait nécessaire de faire une passe supplémentaire sur le code assembleur. Lors de cette passe, le code assembleur dit "mort" doit être donc supprimé.

Techniquement, cette optimisation ne permet pas de réduire le nombre de cycles du programme en question car l'expression `if(false)` sera toujours évaluée et donc le branchement effectué. Néanmoins, si la portion de code est fréquemment réutilisée dans le programme, le fichier assembleur sera surchargé de lignes inutiles. Sa taille sera donc plus importante et l'espace de stockage devra être plus conséquent.

2.2.2 Conditions booléennes

De cette première optimisation en découle une seconde sur les condition booléennes. En effet, lorsque la condition booléenne est composée uniquement de constantes (`true` ou `false`), celle-ci peut être directement évaluée lors de la compilation pendant l'étape de génération de code. Les valeurs de ces constantes étant connues et exploitables par le langage Java, il faut donc évaluer l'ensemble de l'expression contenue dans la condition booléenne.

Un branchement effectué lors d'une instruction conditionnelle consomme au plus cinq cycles sur une machine abstraite IMA. Les instructions LOAD et CMP consomment chacune deux cycles. Si l'instruction conditionnelle contient uniquement des constantes, il est donc possible d'évaluer l'expression et d'éviter l'ensemble de ces instructions. Le programme en ressort alors plus efficient.

```
{
   if (false || true) {
      println("foo");
   }
}
```

Par exemple, dans le programme ci-dessus, l'évaluation de l'expression `(false || true)` serait à la charge du compilateur avant même de générer le code assembleur. Sans l'optimisation en question, le programme assembleur généré actuellement est le suivant :

```
; Main instructions
  LOAD #0, R0
  CMP #0, R0
  BNE else.2_fin
  LOAD #1, R0
  CMP #0, R0
  BEQ else
else.2_fin:
  WSTR "foo"
  WNL
  BRA end
else:
end:
```

En revanche, une fois l'optimisation mise en place, le fichier assembleur correspondant serait :

```
; Main instructions
  WSTR "foo"
  WNL
```

Ainsi, pour cette simple portion de code, les nombres de cycles économisés est le suivant :

$$2 \times \text{LOAD} + 2 \times \text{CMP} + (\text{BNE} \mid \text{BEQ}) + \text{BRA} = 2 \times 2 + 2 \times 2 + 5 + 5 = 18$$

2.2.3 Expressions arithmétiques

Enfin, une dernière optimisation possible s'inspire de la précédente mais pour les opérations arithmétiques. Si ces dernières sont composées uniquement de constantes, l'expression peut être évaluée directement par le compilateur lui-même juste avant la génération de code. Dans ce cas, sont considérées comme constantes l'ensemble des flottants et des entiers. Il est possible de récupérer la valeur des ces constantes grâce à la méthode `getValue()` présente dans les classes `IntLiteral` et `FloatLiteral`. Une fois les valeurs récupérées pour chacun des membres de l'expression, cette dernière peut être donc évaluée dans le code Java du compilateur.

Grâce aux données du polycopié, nous savons que dans une opération arithmétique, une division nécessite quarante cycles internes lors de l'exécution sur une machine abstraite IMA. Une fois de plus, si cette opération venait à être répété un grand nombre de fois dans le programme Deca alors le nombre de cycles internes augmenterait de façon drastique. L'optimisation permet donc d'économiser un certain nombre de cycles internes rendant alors le programme plus efficient et, par conséquent, moins consommateur d'énergie.

2.3 Mesures

Le compilateur `decac` dispose d'une option `-n` (`no check`) permettant à l'utilisateur de spécifier s'il souhaite ou non supprimés certains tests à l'exécution. Ces tests concernent la division par zéro ou encore le débordement de mémoire. La liste exhaustive des tests concernés est disponible dans la section [Sémantique] du polycopié.

L'absence de ces vérifications permet, pour un utilisateur averti, d'éviter un grand nombre d'opérations assembleur. L'efficience du programme concerné est donc optimisée. Le programme devient moins coûteux en énergie car il effectue moins de cycles internes.

Pour vérifier cela, nous avons fait des relevés de performance sur la compilation d'un programme avec et sans l'option `-n`. Soit le programme Deca suivant :

```
{
   int x;
   x = readInt();
   x = x / 10;
   println("x: ", x);
}
```

Sans l'option `-n`, nous obtenons les résultats suivants :

Nombre d'instructions : 25 Temps d'exécution : 167

Avec l'option `-n`, nous obtenons les résultats suivants :

Nombre d'instructions : 13 Temps d'exécution : 149

Ces résultats, obtenus avec l'option `-s` de la commande `ima`, contiennent le nombre d'instructions assembleur et le nombre de cycles internes, correspondant au temps d'exécution. Ainsi, l'option `no check` permet un gain de cycles internes à hauteur de 11%.

3 Procédé de fabrication

Le procédé de fabrication est confronté à deux caractéristiques qui ne sont, à première vue, pas compatibles : respecter la qualité du compilateur et optimiser le procédé de fabrication afin de le rendre plus économe en énergie. Néanmoins, comme nous allons le voir dans les prochaines lignes, nous avons adapté notre processus de validation afin de répondre aux deux spécifications.

Les tests, quelle que soit leur nature (*oracle*, *black-box* ...), sont exécutés par des scripts Shell. Ces derniers traitent, en fonction de la nature du test, la sortie produite par celui-ci afin d'en tirer une conclusion : le test a échoué ou réussi.

En développant une quantité impo300 Wh à raison de 8 heurtante de tests, nous avons rapidement pris de conscience de l'impact énergétique de l'exécution de ceux-ci. Pour palier à cela, une stratégie a été mise en place afin de ne pas à avoir à lancer l'ensemble des tests pour chaque ajout de nouvelles fonctionnalités. Hormis dans de rares cas, notamment pour les rendus intermédiaires, nous n'avons que très peu utilisé la commande `mvn test`.

Ainsi, pour chacune des étapes réalisées par le compilateur, des scripts Shell ont été développés afin d'exécuter les tests utiles pour la partie concernée. Des scripts propres a chacun des éléments suivant ont été développés :

- *Lexer* (`text-lex.sh`)
- *Parser* (`text-pars.sh`)
- *Analyse contextuelle* (`text-context.sh`)
- *Génération de code* (`text-codegen.sh`)
- *Commande* `decac` (`text-decac(2).sh`)

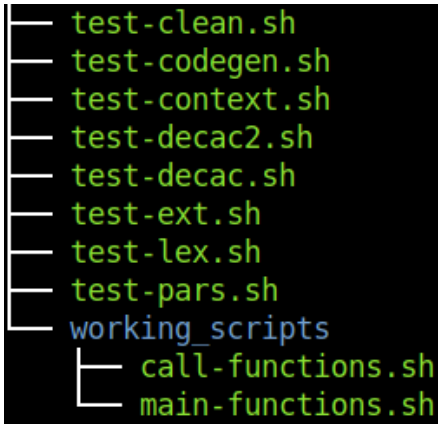

Figure 1 – Arborescence des scripts de tests

Comme nous pouvons le constater dans l'arborescence ci-dessus, des scripts génériques sont présents dans le répertoire `working_scripts/`. Ceux-ci comportent des fonctions permettant de rapidement spécifier le répertoire à tester ainsi que la nature des tests. Les scripts sont donc totalement modulables afin d'être le plus efficient possible. Le fichier `test-ext.sh` est spécifique à l'extension. Ainsi, pendant le développement de l'extension, nous devions uniquement lancer ce script pour valider le code implémenté. Du fait de la division de ces scripts, des économies d'énergie ont donc été réalisées.

De plus, il est à noter que pendant tout le processus de développement des scripts temporaires ont été mis en place. En effet, quand nous avons commencé le développement du compilateur pour la partie avec objet du langage Deca, nous avons mis en place des scripts spécifiques pour la partie avec objet. Ainsi, des scripts pour l'*Analyse contextuelle* et la *Génération de code* permettaient de lancer les tests uniquement pour la partie avec objet du langage. À l'issue du projet, ces scripts ont été supprimés.

Références

[1] Enhancing Software Engineering Processes towards Sustainable Software Product Design, Markus Dick, Stefan Naumann

[2] Economie d'énergie d'un programme informatique, Wikipedia

[3] *Combien d'électricité consomme un ordinateur ?*, hello watt