Compilateur Decac

Analyse des impacts énergétiques et propositions

TEIMUR ABU ZAKI, ADRIEN BOUCHET, TROY FAU, PAUL MARTHELOT, HUGO ROBERT

1 Efficience du code fourni

1.1 Théorie et mesures simples

Dans cette sous-section, nous allons mesurer les temps d'exécution donné par `ima -s` pour chacun des programmes qui seront décrits. Pour les cas les plus simples, nous comparerons cela au nombre de cycles qu'ils seraient , en théorie, possible de réaliser pour obtenir les mêmes résultats.

Dans notre cas, nous nous plaçons dans une configuration où le résultat d'une expression vient d'être calculé dans R0.

1.1.1 Programme vide

Temps d'exécution IMA : 39

Il est facile de remarquer que même pour un programme vide, nous avons déjà beaucoup de cycles d'exécution. Cela s'explique car tout programme produit par notre compilateur commence par un test de dépassement de pile et par la création de la table des méthodes pour Object.

Ces 39 cycles constituent donc une borne inférieur pour le code exécuté. En théorie, sans aucune vérification, nous pourrions arriver à 1 cycle : celui de `HALT`. **Pour la suite de l'analyse, nous soustraierons ces 39 cycles qui ne devraient pas nous intéresser pour les opérations principales.**

1.1.2 Opérations unaires simples

Temps d'exécution IMA : 8 (opposé, NOT true) / 9 (NOT false)

Ces programmes sont de la forme `{ unary_op` *`NumLiteral`* `;}` .

Si le code de l'opération NOT semble bien géré pour un booléen (7 ou 8 cycles), cela ne semble pas être le cas pour l'opérateur Opposé qui est effectué en 8 cycles.

```
;  ------------------------------------------------
;              Main Program
;  ------------------------------------------------
        LOAD #16, R0
        OPP R0, R1
        LOAD R1, R0
        HALT
```

Il serait possible d'améliorer cette opération en faisant tous les calculs dans R0. Cela nous porterait à 6 cycles.

1.1.3 Opérations binaires simples

Temps d'exécution IMA : 12 (+), 13 (-), 30 (x), 62 (/ et %)

Ces programmes sont de la forme `{ NumLiteral binary_op NumLiteral ; }` .

Dans l'idéal, il suffirait de charger l'un des immédiats dans le registre R0 puis d'effectuer l'opération avec l'autre immédiat.

Cependant, nous ne le faisons pas car nous sauvegardons en permanence le résultat du terme gauche dans un registre et nous effectuons le calcul ensuite. Si nous connaissions le type d'expression du terme de gauche, nous pourrions éviter cette sauvegarde inutile et gagner 6 cycles sur ce genre de calculs rapides (soit plus de la moitié des cycles de l'addition).

```
;  -------------------------------------------------
;              Main Program
;  -------------------------------------------------
        LOAD #1, R0
        LOAD R0, R2
        LOAD #1, R0
        ADD R2, R0
        HALT
```

1.1.4 Déclaration et assignation des types prédéfinis

Temps d'exécution IMA : 10

Ces programmes sont de la forme `{ type x = Literal ; }` .

Ce résultat est le fruit d'un `LOAD` d'immédiat puis d'un `STORE` dans la pile. Il semble difficile de faire mieux à moins d'adopter d'autres moyens de sauvegarder les variables (par exemple dans les registres).

1.1.5 Branchements conditionnels

Temps d'exécution IMA : 21

Ces programmes sont de la forme `{ if (!true) {} else if (true) {} else {} }` .

Si nous n'avons pas trouvé comment améliorer ce nombre de cycles pour des évaluations simples de condition, il semblerait que nous puissions améliorer l'évaluation des expressions élaborées en utilisant l'évaluation paresseuse.

Pour l'instant, ce n'est pas le cas, nous pouvons d'ailleurs le voir ici dans un exemple où la première condition est `(false && true)`.

Pour le *While*, nous n'avons pas identifié de problème particulier et ne voyons pas comment l'améliorer.

1.1.6 Déclarations de classes

Temps d'exécution IMA : $20 + 10 \times$ nb méthodes

```
;  ------------------------------------------------
;              Main Program
;  ------------------------------------------------
       LOAD #0, R0
       LOAD R0, R2
       LOAD #1, R0
       ADD R2, R0
       CMP #2, R0
       SEQ R0
       CMP #1, R0
       BNE label0
       BRA label1
```

Nous n'avons pas trouvé de moyens de passer en-dessous de ce seuil car la technique donnée dans les spécifications semble déjà optimisée pour respecter les conventions.

1.1.7 Instanciations de classes

Temps d'exécution IMA : $112 + 32 \times$ nb de champs

Idem que pour la déclaration des classes.

1.1.8 Instanceof

Temps d'exécution IMA : $34 + 27 \times$ profondeur

A ce niveau, nous n'avons pas réussi à identifier de pistes d'améliorations.

1.2 Comparatifs avec les autres groupes sur les tests

La sous-section précdente permet d'identifier quelques points d'améliorations au niveau de l'implémentation locale des instructions.

Regardons maintenant notre écart avec les autres groupes du palmarès (à la date du 25/1/2022) pour vérifier si nos améliorations paraissent nécessaires ou non. En particulier, nous nous intéresserons aux groupe 38 : groupe le plus fort sans avoir l'extension OPTIM.

Si notre nombre de cycle est noté c et le leur c_{min}. L'écart relatif sera calculé comme :

$$E = \frac{|c - c_{min}|}{c}$$

Groupe	GL49 (nous)	GL38	Écart Relatif (%)
syracuse42	1782	1346	24,5
ln2	18322	11402	37,7
ln2_fct	59538	33482	43,7

1.3 Bilan de l'efficience du code généré

Au vu de ces résultats, nous pouvons comprendre plusieurs choses :

— Les optimisations au niveau des opérations binaires évoquées plus haut, pourraient avoir comblé jusqu'à 10% au moins de l'écart puisque la majorité des opérations sont des additions (optimisables de 8 à 6 cycles, i.e. 25%) dans le test.

— Au fil des sous-langages, l'écart entre les meilleurs et nous s'est creusé. Cela est sûrement dû à l'accumulation d'opérations non-optimales de notre côté.

2 Efficience du procédé de fabrication

Les chiffres qui vont suivre ont été estimés par la moyenne de 5 résultats obtenus à l'issue de la commande `/usr/bin/time -v` *`commande`*. Nous considérons le temps utilisateur + système sans vraiment s'intéresser au taux d'utilisation du CPU.

La machine utilisée a pour propriétés :

— Système d'exploitation : Ubuntu 20.04.3 LTS, 64 bit

— Processeur : Intel® Core™ i5-8265U CPU @ 1.60GHz × 8

— Mémoire : 7.6 GiB

Nous utiliserons le TDP (*Thermo Design Power*) de $15W$ de ce CPU pour convertir le temps de calcul en énergie. Cette unité de mesure est très discutable car c'est une borne supérieure : l'ordinateur ne fonctionne jamais à plein régime. Cependant, cela devrait permettre de parer aux éventuelles sous-estimations de répétitions des commandes.

2.1 Coût des commandes fréquentes

Lors du projet, nous avons identifié plusieurs commandes très gourmandes en énergie mais dont nous avons souvent eu besoin.

Le tableau ci-dessous récapitule alors ces commandes et le temps passé à leur exécution (nous nous plaçons dans le pire cas où fichiers que nous avons écrits sont inclus) :

Commande mvn	Temps moyen (s)	Répétitions	Temps total estimé (s)
compile	17,33	500	8665
test-compile	19,42	100	1942
verify	1037,11	70	72598
		Total	83204

Au total, nous avons donc passé environ 83000s à utiliser l'ordinateur, ce qui vaut correspond à plus de 23h d'utilisation intensive.

Les autres commandes maven ont rarement été utilisées et nous estimons que leur impact est négligeable devant les autres.

2.2 Coût de la commande `decac`

Si on compte les commandes `decac` utilisées en dehors de `mvn verify`, nous avons estimé à plus de 10 000 le nombre de lancements sur les fichiers. (Nous avions près de 940 tests, il ne paraît pas aberrant de les avoir compilé, manuellement ou à travers un script, une dizaine de fois chacun)

On peut distinguer deux types de fichiers Deca :

— ceux qui testaient les dépassements de pile ou de tas : il y en a 3 et prennent en moyenne 20,77s à être compilés ensemble.

— les autres : nous en avons pioché une trentaine au hasard et avons calculé la moyenne du temps de compilation. On obtient alors 0,51s en moyenne (et un écart-type de l'ordre de 0,05s).

Au final, on peut estimer à $(20,77 + 0,51 \times 940) \times 10 = 5001,7$s le temps d'exécution de la commande `decac` tout au long du projet

2.3 Coût de la commande `ima`

Nous n'avons utilisé la machine virtuelle `ima` que sur les tests de génération de code. Nous estimons l'avoir fait un millier de fois.

Or, en lançant plusieurs tests, nous nous sommes rendus compte que le temps passé sur cette commande est négligeable pour nos exécutables (moins de $0,01$s).

Nous ne compterons donc pas cela dans notre bilan.

2.4 Coût de la commande `ARM-exec.sh`

Á l'instar de la commande `ima`, nous utilisions un script basé sur compilateur-croisé pour pouvoir exécuter les tests en assembleur à destination de ARM.

Nous estimons avoir lancé manuellement une centaine de fois la commande `ARM-exec.sh` En moyenne sur 25 tests pris au hasard, on trouve un temps d'exécution de $0,09s$.

Nous évaluons donc à $0,09 \times 100 = 9s$ le temps de fonctionnement du CPU sur cette commande. C'est assez négligeables par rapport au reste des commandes.

2.5 Bilan de l'efficience du procédé de fabrication

En sommant les résultats de la sous-section précédente, on trouve un temps d'exécution total de l'ordre de 89000 s pour le CPU, soit 24h30, ce qui donne une borne supérieure de $1,335$MJ $= 0,371$kWh utilisés au maximum. Ce chiffre est assez cohérent lorsqu'on regarde la consommation électrique journalière d'un ordinateur portable (cf. 3.1.2) qui a le même ordre de grandeur.

En utilisant l'outil de conversion présent sur `https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator`, on trouve que cela est équivalent à $0,262$kg d'émissions de CO_2.

Pour mettre cela en perspective, cela est comparable à près d'un 1 km de voiture ou à la recharge de 32 portables.

Nous nous apercevons donc que les procédés de fabrication ont un impact assez fort sur l'environnement. C'est d'autant plus vicieux que nous ne le ressentons pas vraiment au cours du développement. Toutefois, c'est moins impressionant que ce à quoi nous nous attendions.

De plus, nous identifions que la majorité des ressources ($\sim 95\%$) a été consommée par les commandes maven et en particulier la commande `mvn verify` qui a consommé quatre cinquième des ressources à elle seule.

Cependant, ce risque était déjà marqué dans notre esprit au début du projet et nous avons donc pu prévenir ce problème dès le début.

2.6 Actions mises en places pour éviter la sur-consommation d'énergie

- **Conception granulaire des tests :** Comme vous pourrez le trouver dans le document de validation, nous avons conçu des scripts permettant de cibler des tests pour des sous-langages particuliers. Cela nous permettait d'éviter une surconsommation due aux tests inutiles au sous-langage testé.
- **Lancement modéré de `mvn verify` :** Dans les deux premières semaines, nous n'avons pas lancé `mvn verify` car nous devions faire en sorte de nettoyer les tests unitaires.

  Ensuite, lors de la troisième semaine, au moment de commencer la partie Objet, nous avons commencé à la lancer mais en se limitant à une ou deux fois par jour.

  Cependant, lors du dernier week-end, nous avons lancé plus d'une cinquantaine de fois le script afin d'être sûr que chacune de nos modifications pour débogages n'engendraient pas de régression.
- **Génération modérée de tests automatiques :** Dans les parties A et B, il pouvait être tentant de générer une multitude de tests pouvant couvrir tout l'arbre de syntaxe abstraite.

  Or, nous nous sommes rendus compte que la plupart des tests pouvaient être redondants et ne faisaient que rallonger les scripts de validation automatique.

  Nous avons donc limités le nombre de tests générés automatiquement.

3 Autres impacts écologiques

Il faut également penser qu'afin de travailler, nous avons utilisé des outils. Or ces outils consomment également des ressources énergétiques. Nous nous tentons alors à l'évaluation de ces impacts qui sont les plus difficiles à analyser.

3.1 Postes de dépenses identifiés

3.1.1 Déplacements

4 des membres de l'équipe traversaient la ville en tramway pour rejoindre l'Ensimag.

Bien que cela semble être un poste de dépense incompressible car notre gestion de projet a demandé la présence de chacun des membres pour le travail en commun, nous aurions probablement pu travailler depuis chez nous.

Par exemple, nous aurions pu discuter en utilisant des services comme Discord. Cependant, nous manquons d'informations sur la réelle économie de ressources qui aurait pu être faite.

3.1.2 Mise sous tension des ordinateurs

Parmi les cinq membres de l'équipe, seuls 2 travaillaient vraiment sous envrionnement Linux. Les autres utilisaient principalement des machines virtuelles. Lors des phases de développement, nous avons pu vivre la lenteur ou la consommation de ces machines virtuelles.

Cependant, nous avons cherché à limiter la consommation totale en ne travaillant que sur nos ordinateurs portables (seul l'un d'entre nous travaillait sur les PC de l'Ensimag).

On peut notamment voir cet écart de consommation dans la figure suivante :

kWh	par jour	par mois	par an
ordinateur portable (type Notebook)	0.20 kWh	6.25 kWh	75 kWh
ordinateur de bureau (type Mac intégré)	0.538 kWh	17.5 kWh	210 kWh
ordinateur pc	0.766 kWh	23.3 kWh	280 kWh

Comparaison des consommations de différents types d'ordinateurs

Source : https://plum.fr/blog/astuces-eco-gestes/consommation-electrique-ordinateur/

3.1.3 Recherches Internet

Lors de la phase de recherche sur l'extension pour la génération de code à destination de machines ARM, nous avons pris au moins trois jours à trouver un ensemble d'instructions, un FPU et une architecture compatibles.

Au cours de ces trois jours, nous avons dû charger une centaine de pages Internet, principalement hébergées des deux côtés de l'Atlantique.

Nous n'avons pas vraiment réussi à évaluer cet impact.

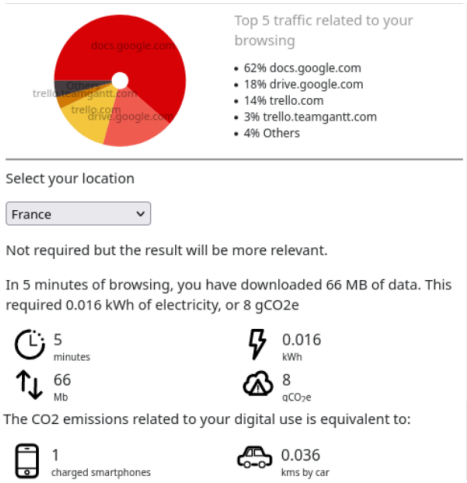
3.1.4 Utilisation d'un Drive, de Discord et de Trello en plus de GitLab

Bien qu'ayant à disposition un GitLab hébergé par l'Ensimag, nous avons décidé de partager un Drive et d'établir nos plannings sur Trello. En effet, ces deux outils permettent une meilleure utilisation collective tout en proposant des services plus puissants.

De plus, nous avons utilisé un serveur Discord pour communiquer à l'écrit pour le travail nocturne.

Nous n'avons pas réussi à évaluer l'impact des ces services en ligne tout au long du projet, mais nous savons pour sûr que GitLab est de loin la solution la moins impactante.

En effet, l'un des membres a lancé une analyse des impacts relatifs de nos requêtes Internet pour le GitLab, le Drive et Trello grâce au service Carbonalyser.



3.2 Bilan des impacts numériques indirects

Comme vous aurez pu le constater, nous manquons de données fiables pour évaluer notre impact écologique indirect.

Cependant, en utilisant les ressources de l'INR et de Décathlon, nous avons essayé de calculer notre impact énergétique en termes de CO_2 produit (sur 3 semaines sans compter les coûts de production de nos ordinateurs).

Vous pourrez retrouver ce calculateur ici : `https://institutnr.org/calculatrice/impact-environnemental-numerique-inr.html`

Cependant, cette évaluation est assez grossière car elle ne prend pas en compte les coûts de recherches Internet et de mise en tension des ordinateurs.

— Impact numérique (stockage Drive et mails) : 0,23 kg eq CO2 (quasiment ce qu'on trouve pour les procédés de fabrication) ;
— Impact déplacements : 12,82 kg eq CO2 (soit plus de 50 fois l'impact numérique) ;
— **Impact total : 13,05 kg eq CO2**

L'impact déplacements représente à lui seul 1680 recharges de téléphone et plus de 50km en voiture. L'impact numérique en devient dérisoire même s'il représente quasiment autant que les procédés de fabrication.

Nous pourrions donc légitimement nous demander s'il est vraiment nécessaire de chercher à optimiser nos produits à des fins écologiques puisqu'en théorie, le simple fait de se déplacer en véhicule dépassera l'exécution de milliers de programmes Deca.

Plutôt que d'y réfléchir d'une manière écologique, il faudrait voir l'efficience des programmes générés comme une course à l'accélération des programmes.

4 Voir au-delà du projet : qu'avons-nous réalisé ?

Ce document d'analyse énergétique a sûrement été demandé pour initier les élèves à la Responsabilité Sociétale des Entreprises.

Or, la démarche de RSE s'inspire de la philosophie de René Dubos : « Agir local, penser global ». Si le projet GL nous invite bien à revoir l'efficience et les impacts écologiques de notre projet, nous pensons rarement à ce qui se passera après ; c'est là, nous le remarquons, un manquement à la directive « penser global ».

Mettons donc en perspective les ressources consommées et les ressources qui seront économisées. Avons-nous répondu à un besoin existant ou amélioré la solution pré-existante à ce besoin ?

Nous exclurons d'ailleurs le besoin éducatif car le projet Génie Logiciel aurait pu se étudier un autre produit qu'un compilateur.

4.1 Avons-nous répondu à un besoin ?

Oui mais notre client est fictif, son langage Deca et ses besoins encore plus. En réalité, de nombreux compilateurs peu gourmands pour des langages de programmation Orientées Objet existent déjà sur le marché.

4.2 Utilisation ultérieures de notre compilateur ?

Notre projet est motivé par la création d'un compilateur pour un langage artificiel de programmation orienté objet. Il ne sera donc peu (et vraisemblablement pas du tout) utilisé par des programmeurs dans le futur. En revanche, quelques tests de rapidité faits sur de petits programmes Deca et Java montrent que non seulement Decac est plus rapide que Javac mais aussi que IMA consomme moins que la JVM. Cela est très intéressant, d'autant plus que la syntaxe du langage est plutôt complète,

très simple et intuitive.

Autre point notable : même si cet exercice de gestion de projet est important dans la vie d'un ingénieur (ou d'un chercheur), il est fortement probable qu'une infime partie des élèves de notre promotion ne vienne jamais à écrire un compilateur et donc ne mette ce savoir à disposition d'un futur projet d'entreprises.

4.3 Impact écologique de Decac ?

Comme nous l'avons vu précédemment, notre efficience de code et de procédés de fabrications sont bien peu de choses face à nos impacts indirects de déplacements. Nous ne pouvons donc pas consciemment dire que notre compilateur va changer le cours des choses.

4.4 Conclusion

Même si notre compilateur est en quelque sorte une bonne introduction aux prouesses techniques possibles en Génie Logiciel, nous ne pouvons pas nous targuer d'avoir créé un produit très utile.
C'est assez dommage même le produit est à la charge du client. Nous pouvons seulement espérer du langage Deca qu'il soit utilisé comme un gadget, à l'instar de la calculatrice ordinateur, et que ce jour-là Decac soit utilisé.