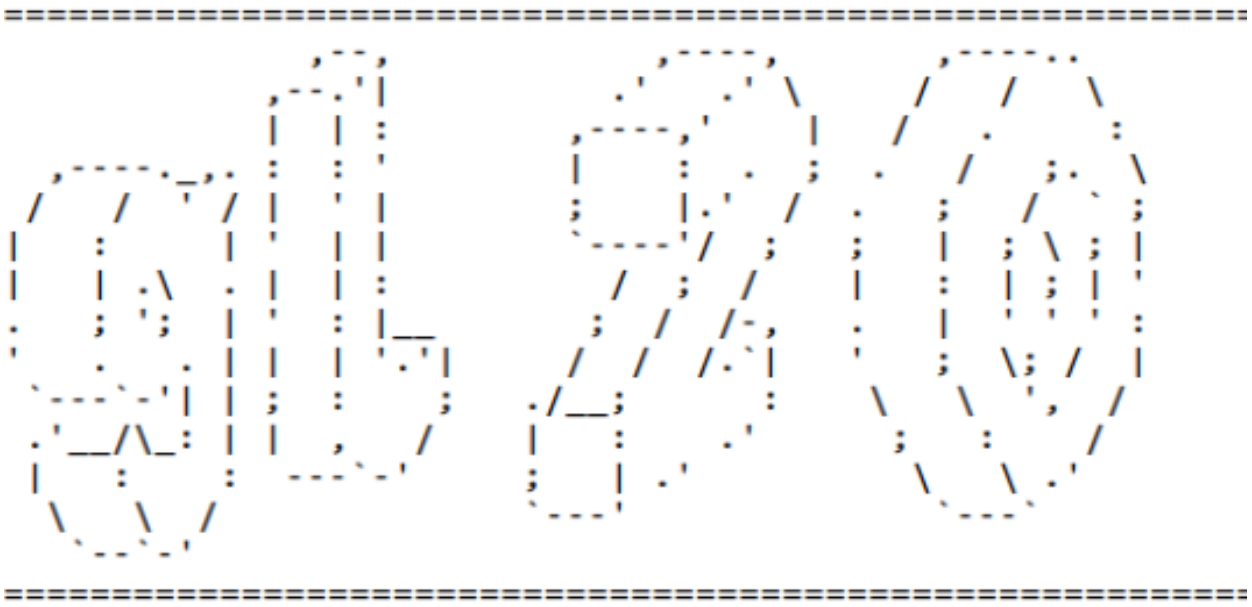
Analyse des impacts énergétiques du Projet

- Groupe GL20



Melvin Bonamour
Steve Lening
Mathieu Meunier
Paul Grassi
Emmanuel Loroux

Table des matières

I - Méthode de développement

II - Coût énergétique du projet

II.1 - Le coût de l’exécution

II.2 - Coût de fabrication du compilateur

II.3 - Estimation du coût total de fabrication

III - Conclusion

Annexe

I - Méthode de développement

Concernant le développement, nos recherches nous ont appris que la consommation énergétique d'un ordinateur portable était de 50 à 80 pourcents inférieure à celle d'un ordinateur de bureau. Nous avons donc décidé d'utiliser quasiment exclusivement nos ordinateurs portables pour développer notre projet.

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

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

Cependant, bien qu'il aurait été préférable de travailler exclusivement en distanciel, pour des raisons d'efficacité, nous avons décidé de travailler majoritairement en présentiel pour favoriser l'échange facile et rapide d'informations et booster la motivation ainsi que la productivité des membres de notre équipe. Nous avons toutefois privilégié les moyens de transport les plus écoresponsables pour se rendre à notre lieu de travail (l'ENSIMAG). Nous nous sommes donc quasiment exclusivement déplacés soit à vélos, soit en tramway.

II - Coût énergétique du projet

Pour évaluer la consommation énergétique du projet, nous nous sommes intéressés principalement à deux aspects :

II.1 - Le coût de l'exécution

Nous avons estimé ce coût grâce deux indicateurs :

- La durée d'exécution.
- Le coût du code assembleur généré (nombre de cycles)

La durée d'exécution du compilateur est fonction de la complexité du fichier source deca. C'est pour cela que nous allons partir d'un code source deca spécifique pour

évaluer la durée d'exécution et le coût en nombre de cycles du code assembleur de ce dernier. Voici le code deca que nous considérons :

```
class C {
        int x;
        int y=1;
        int getX() {
                return x;
        }
        void incrX() {
                x = x + 1;
        }
}
{
        C c = new C();
        c.x = 2;
        print(c.x);
}
```

Il s'agit d'un exemple tiré du poly-projet-GL section [GenCode] partie 4, page 214.

Nous travaillons actuellement sur un PC Core I3, 8ième génération qui dispose de 12Go de RAM. Le Système d'Exploitation utilisé est Ubuntu 22.04.

- Durée d'exécution :

  Pour exécuter le compilateur, il faut tout d'abord effectuer la commande **mvn compile** pour compiler le projet (cela prend environ **10s**), puis lancer la commande **decac [option] [nom_du_fichier_source]**, vous pouvez vous référez à la documentation utilisateur pour avoir plus de détails sur l'utilisation de la commande decac.

  La durée d'exécution totale du code source deca précédent est : **10,56 s.**

  Dans les prochaines exécutions, on n'aura plus besoin de faire mvn compile et ainsi on pourra compiler ce code en **56 ms**.

- Coût du code assembleur généré (en nombre de cycles internes) :

  Nous évaluons ce coût en comptabilisant le nombre de cycles du code assembleur généré. Pour cela on se base sur le nombre de cycles internes

des instructions ci-dessous fournies dans le poly section [MachineAbstraite] partie 3.8, page 110.

LOAD 2 DIV 40

STORE 2 INT 4

LEA 0 BRA 5

PEA 4 Bcc 5 (cc vrai) 4 (cc faux)

PUSH 4 BSR 9

POP 2 RTS 8

NEW 16 DEL 16

ADD 2 RINT 16

SUB 2 RFLOAT 16

SHL 2 WINT 16

SHR 2 WFLOAT et WFLOATX 16

OPP 2 RUTF8 et WUTF8 16

MUL 20 WSTR 16

CMP 2 WNL 14

QUO 40 ADDSP 4

REM 40 SUBSP 4

FLOAT 4 TSTO 4

Scc 3 (cc vrai) 2 (cc faux) HALT 1

ERROR 1 SCLK 2

SETROUND_mode 20 CLK 16

FMA 21

Partant de ces informations, nous constatons que le code assembleur résultant du code source deca précédent effectue **326 cycles** (Le code assembleur généré est donné en Annexe de ce document).

En ce qui concerne la partie C du compilateur Deca, nous avons adopté certaines pratiques qui nous ont permis de réduire le nombre de cycles internes du code assembleur. Une de ces pratiques a consisté à utiliser une seule instruction TSTO par bloc - par bloc, nous entendons une méthode, un sous-programme d'initialisation des champs d'une classe ou le programme principal - plutôt que de mettre une instruction TSTO à chaque fois qu'on a besoin d'ajouter un élément dans la pile. Nous sommes néanmoins conscients qu'on aurait pu réduire davantage les dépenses énergétiques

dûes à l'exécution de notre compilateur en améliorant par exemple la qualité du code assembleur généré, c'est-à-dire en faisant une passe supplémentaire à l'étape C qui permettrait de supprimer les instructions assembleurs superflues, ce qui permettrait de réduire de façon significative le nombre de cycles du code assembleur généré.

II.2 - Coût de fabrication du compilateur

Nous pouvons séparer le coût de fabrication de notre compilateur comme suit :

- Le coût de la compilation.
- Le coût de l'exécution des tests.

- Coût de la compilation :

Ici, la méthode idéale que nous aurions souhaité adopter serait de minimiser le nombre d'appels de la commande mvn compile, et de limiter le nombre de machines qui feraient des tests. Ainsi, on ne lancerait une compilation que lorsqu'une fonctionnalité aurait été développée dans son entièreté et récupérant le plus de modifications possibles avant la compilation. Cependant, dans l'exécution nous avons remarqué que cette méthode serait risquée et assez incompatible avec notre méthode de travail car chaque fonctionnalité était développée sur git sur une branche isolée et que ce n'était qu'une fois assurés du bon fonctionnement de cette fonctionnalité qu'elle était intégrée à l'ensemble du projet. Toutefois c'était bien en général à la fin du développement de la fonctionnalité dans son entièreté qu'une compilation était lancée et que les tests propres à cette fonctionnalité (Voir section suivante - *Méthode de test*) étaient lancés.

- Méthode de test :

Ici, la méthode de test retenue était de lancer un ensemble de tests propres à une fonctionnalité une fois le développement de celle-ci terminée. Les tests étant lancés à cette phase étant uniquement les tests propres à cette fonctionnalité et non ceux qui avaient été créés pour d'autres fonctionnalités. Par exemple, pour l'affichage des chaînes de caractères, les tests lancés seraient la compilation d'un programme vide, d'un programme contenant un bloc vide, un programme une seule instruction d'affichage d'une chaîne de caractère et un programme contenant une instruction d'affichage d'un ensemble de chaîne de caractères car une instruction **print ou println** peut prendre plusieurs arguments.

Ces tests, une fois tous passés ne seraient plus relancés au développement d'une autre fonctionnalité. Les tests unitaires étaient donc lancés au cours de la journée à chaque ajout de fonctionnalités.
Au terme de la journée, l'ensemble du code avec toutes les fonctionnalités ajoutées depuis le début du projet était recompilé et cette fois-ci testé avec l'ensemble des tests conçus depuis le début du développement. Ainsi, toutes les fonctionnalités étaient re testées et en cas de bugs corrigées soit dans la nuit, soit le jour suivant.
Ce mode de fonctionnement permettrait de réduire la consommation énergétique en réduisant le nombre de compilations et de tests, sans perdre en efficacité et en qualité du compilateur. En effet, bien que nous aurions pu grandement réduire le coût énergétique du projet en se limitant uniquement au tests propres à chaque fonctionnalités à chaque ajout de fonctionnalité, ce fonctionnement aurait présenté des risques très importants au niveau du fonctionnement du compilateur, car une fonctionnalité nouvellement ajoutée au projet aurait pû causer des problèmes/dysfonctionnements à une autre déjà implémentée sans qu'on ne s'en rende compte (Ces test globaux constituent donc ce que l'on appelle **tests de régression** qui permettent de se rassurer que les fonctionnalités ajoutées sont en adéquation avec ce qui était la précédemment, en claire, ce qui marchait avant l'ajout de cette fonctionnalité doit toujours être fonctionnel).

II.3 – Estimation du coût total de fabrication

Notre équipe compte quatre ordinateurs PC et un ordinateur portable type NoteBook, en se basant sur la consommation moyenne des ordinateurs portables fournie dans la partie I de ce document, et étant donné que le projet s'est déroulé sur 03 semaines et 04 jours (Nous n'avons pas travaillé sur le projet tous les jours pendant les congés de Noel, juste 04 jours), la consommation énergétique liée à la fabrication de notre compilateur est d'environ **81.6 kWh** en tout.

III - Conclusion

Parvenu au terme de ce projet, nous avons pu constater qu'il était assez difficile de prendre en compte à la fois les dépenses énergétiques liées à un projet et le souci de qualité du produit obtenu car on se concentre généralement sur la qualité en reléguant les impacts énergétiques au second plan. Nous avons compris tout au

long de ce projet les enjeux de l'impact énergétique d'un projet et en avons fait une de nos priorités dès le début du projet. Nous avons ainsi pu adopter tout au long du projet des pratiques qui permettraient de minimiser l'impact énergétique du projet. Il apparaît alors clairement que l'impact énergétique d'un projet quelle que soit sa taille est devenu un facteur à prendre très au sérieux et nous comptons bien prendre en compte cet aspect dans nos futurs projets académiques et professionnels.

Annexe :

```
; start main program
    TSTO #7
    BOV stack_overflow_error
    ADDSP #8
; Code de la table des méthodes de Object
    LOAD #null, R0
    STORE R0, 1(GB)
    LOAD code.Object.equals, R0
    STORE R0, 2(GB)
; Code de la table des méthodes de C
    LEA 1(GB), R0
    STORE R0, 3(GB)
    LOAD code.Object.equals, R0
    STORE R0, 4(GB)
    LOAD code.C.getX, R0
    STORE R0, 5(GB)
    LOAD code.C.incrX, R0
    STORE R0, 6(GB)
; Main program
; Beginning of main instructions:
    NEW #3, R2
    BOV heap_error
    LEA 3(GB), R0
    STORE R0, 0(R2)
    PUSH R2
    BSR init.C
    POP R2
    STORE R2, 7(GB)
    LOAD 7(GB), R2
    CMP #null, R2
    BEQ dereferencement_null
    LOAD #2, R3
```

```
    STORE R3, 1(R2)
    LOAD 7(GB), R2
    CMP #null, R2
    BEQ dereferencement_null
    LOAD 1(R2), R2
    LOAD R2, R1
    WINT
    HALT
; end main program
; Initialisation des champs de C
init.C:
    TSTO #1
    BOV stack_overflow_error
    LOAD -2(LB), R1
; Sauvegarde des registres
    PUSH R2
; Initialisation a 0 de x
    LOAD #0, R0
    LOAD -2(LB), R1
    STORE R0, 1(R1)
; Initialisation a 0 de y
    LOAD #0, R0
    LOAD -2(LB), R1
    STORE R0, 2(R1)
; Initialisation explicite de y
    LOAD #1, R2
    LOAD R2, R0
    LOAD -2(LB), R1
    STORE R0, 2(R1)
; Restauration des registres
    POP R2
    RTS
; Code de la méthode equals
code.Object.equals:
```

```
    LOAD -2(LB), R2
    CMP -3(LB), R2
    BNE not_equals.1
    LOAD #1, R0
    BRA fin_equals.1
not_equals.1:
    LOAD #0, R0
fin_equals.1:
fin.Object.equals:
    RTS
; Code de la méthode getX
code.C.getX:
    TSTO #1
    BOV stack_overflow_error
    ADDSP #0
; Sauvegarde des registres
    PUSH R2
    LOAD -2(LB), R2
    CMP #null, R2
    BEQ dereferencement_null
    LOAD 1(R2), R2
    LOAD R2, R0
    BRA fin.C.getX
    WSTR "Erreur : sortie de la methode C.getX sans return"
    WNL
    ERROR
fin.C.getX:
; Restauration des registres
    POP R2
    RTS
; Code de la méthode incrX
code.C.incrX:
    TSTO #1
    BOV stack_overflow_error
```

```
    ADDSP #0
; Sauvegarde des registres
    PUSH R2
    LOAD -2(LB), R2
    CMP #null, R2
    BEQ dereferencement_null
    LOAD -2(LB), R3
    CMP #null, R3
    BEQ dereferencement_null
    LOAD 1(R3), R3
    LOAD #1, R4
    ADD R4, R3
    BOV overflow_error
    STORE R3, 1(R2)
fin.C.incrX:
; Restauration des registres
    POP R2
    RTS
; Errors messages
overflow_error:
    WSTR "Error: Overflow during arithmetic operation"
    WNL
    ERROR
stack_overflow_error:
    WSTR "Erreur : La pile est pleine"
    WNL
    ERROR
heap_error:
    WSTR "Erreur : Le tas est plein"
    WNL
    ERROR
io_error:
    WSTR "Error: Input/Output error"
    WNL
```

```
    ERROR
dereferencement_null:
    WSTR "Erreur : Dereferencement de null"
    WNL
    ERROR
```