


Projet Génie Logiciel

Analyse d'impact énergétique

ENSIMAG 2ème année - 2022-2023

Groupe 1 - équipe 3

Jorge Luri Vañó
Nils Depuille
Vianney Vouters
Virgile Henry
Logan Willem



I. Introduction	2
II. Moyen d'évaluation	2
1. La consommation du code compilé	2
2. La consommation du compilateur	3
3. La consommation des outils	4
III. Choix de compilation	4
IV. Choix de validation	4
1. Usage de programmes automatiques	5
2. Tests unitaires VS tests globaux	6
V. Extension	7
VI. Conclusion	8



I. Introduction

Le Projet GL est un travail qui demande, non seulement un code fonctionnel, mais aussi un code respectueux de la planète en optimisant sa dépense énergétique. Que ce soit à travers nos pratiques de programmation ainsi que l'efficacité de notre compilateur, nous avons toujours essayé de tous nos moyens de réduire notre dépense énergétique. Nous avons alors établi plusieurs moyens pour mesurer et contrôler notre consommation, ce qui a représenté un des enjeux majeurs de notre travail.

II. Moyen d'évaluation

Avant de pouvoir commencer à limiter notre consommation, nous devons avoir un moyen d'évaluer comment notre code et notre compilateur dépensent en énergie, en essayant de tous nos moyens de sortir des données explicites afin d'avoir un vrai point de comparaison. Pour cela, nous distinguons plusieurs moyens d'évaluation.


1. La consommation du code compilé

Le principe d'un compilateur, comme il est explicite, est de compiler un code. Ce programme compilé peut alors être exporté de manière pas optimale énergétiquement ou en cherchant à consommer le moins possible. Pour cela, nous avons essayé de trouver un moyen de calcul de la consommation du programme compilé.

Un processus prend une quantité définie de cycles à effectuer les tâches pour être complété. Nos programmes étant essentiellement des programmes qui réalisent des calculs, des appels d'instructions utilisateur et des renvois de littéraux, la consommation de la carte graphique n'est quasiment nulle (très négligeable) et les échanges avec la RAM très limités (aussi très négligeables face au reste de la consommation du système). De fait, le travail est réalisé presque intégralement par le processeur.

De plus, le compilateur n'a pas été conçu pour travailler en multicœur sur notre machine. Nous pouvons considérer que le travail est réalisé par un seul cœur du processeur.

Ainsi, la consommation du code compilé peut être connue de manière approximative (mais très proche de la réalité) grâce à la consommation du cœur du processeur par cycle de calcul.



Dans un ordinateur, nous pouvons facilement connaître la Fréquence d'un Processeur ainsi que sa Puissance Nominale (Wattage Rating) du processeur. Le rapport des deux données donne comme résultat la consommation énergétique en joules d'un cycle du processeur. Ainsi, la multiplication du rapport par le nombre de cycles du programme compilé indique comme résultat l'énergie consommée de manière approximative par le programme en joules.

En tout,

$$E = \frac{\text{Puissance Nominale}}{\text{Fréquence du Processeur}} \times (\text{Nombre de cycles})$$

Avec E l'énergie consommée en Joules, la Puissance Nominale en Watts, la Fréquence du Processeur en Hz et le nombre de cycles sans unité. La formule est bien homogène.

Pour nos calculs, nous avons établi comme machine échantillon un ordinateur avec un processeur à 50W de Puissance Nominale et 2.5 GHz de fréquence. Ainsi, pour notre projet, nous avons :

$$E = 2 \cdot 10^{-8} \times (\text{Nombre de cycles})$$

Finalement, pour obtenir le nombre de cycles d'un programme compilé, nous devons exécuter IMA avec l'option -s : cette option renvoie le nombre de cycles utilisés par le programme, ce qui donne l'énergie consommée grâce à la formule établie précédemment.

2. La consommation du compilateur

L'idée de calcul de la consommation énergétique du compilateur revient à réaliser les mêmes calculs que pour le code compilé. Notre compilateur n'est pas non plus programmé pour travailler en multicœur. Nous pouvons alors considérer que le compilateur utilise uniquement un cœur du processeur comme nous avons étudié auparavant. On peut donc utiliser la même formule de calcul que pour le code compilé. Nous devons alors obtenir le nombre de cycles utilisés par le compilateur.

Pour Maven, nous n'avons pas une option -s qui renvoie le nombre de cycles du processeur directement. Ainsi, nous avons utilisé la commande terminale time pour connaître le nombre de cycles de la commande `mvn compile`. Nous pouvons alors calculer l'énergie consommée par le compilateur en joules.

3. La consommation des outils

Pendant notre projet, nous avons utilisé un grand nombre d'outils : Gitlab, Google Docs, Discord... L'énergie consommée par les outils utilisés étant très difficiles à déterminer, nous n'avons pas pu obtenir une valeur numérique de la consommation. Nous avons donc appliqué le bon sens pour essayer de réduire au maximum notre empreinte énergétique.

III. Choix de compilation

Un des problèmes que nous avons eus lors de la réalisation du compilateur, du point de vue énergétique, est que la structure du compilateur, elle est en soit déjà donnée. Ainsi, la partie la plus libre de la partie non-extension du projet c'était la partie C et même dans cette partie, nous sommes assez biaisés pour réaliser ce que le projet veut que nous fassions. Néanmoins, nous avons trouvé plusieurs moyens pour réduire le nombre de cycles du processeur, ce qui a signifié une diminution de la consommation énergétique du compilateur.

De la même manière que la factorisation du code peut optimiser le code exporté, si nous factorisons le code du compilateur, nous pouvons réduire le nombre de cycles utilisés par le compilateur.


Pour cela, nous avons essayé d'appliquer des règles de codage :

- Les méthodes doivent être définies dans les classes les plus générales possibles.
- On a essayé de réduire au maximum les variables déclarées.
- On a réduit les calculs redondants
- On a évité des boucles pas nécessaires.

En tout, nous avons réduit au maximum au niveau du compilateur le nombre de cycles pour compiler un programme.

IV. Choix de validation

Un projet si important comme le Projet GL a besoin évidemment d'une phase de validation réitérée qui teste la fonctionnalité et l'optimalité du code réalisé. Plusieurs manières de validation peuvent entrer en jeu pour attester du bon fonctionnement du code. Néanmoins, nous devons



considérer quelles procédures sont nécessaires et lesquelles sont évitables pour réduire le surplus énergétique.

1. Usage de programmes automatiques

Lorsque nous avons commencé le projet, nous avons voulu mettre en place plusieurs systèmes de validation automatique.

Tout d'abord, nous voulions utiliser les Pipelines Gitlab. Fonctionnant comme un test automatique, les Pipelines permettent à chaque commit de vérifier en temps réel si le code envoyé est correct, passe les différents tests et si l'avancement est dans la bonne direction. Ainsi, nous pourrions voir si le nouveau commit contredit un test validé au préalable ou si des nouveaux tests sont validés. Néanmoins, cela signifie qu'à chaque commit, l'entièreté des tests devraient être exécutée pour contrôler le nouveau code. Nous avons réalisé au total environ 700 commits durant notre projet. Les commits réalisés sont très souvent des résolutions de bugs, ajout de nouveaux tests et ajout de nouvelles fonctionnalités. D'après notre charte d'équipe, le développeur doit tester son code en local avant d'envoyer le commit, donc le test est déjà testé en amont de réaliser le commit. Ainsi, même si les Pipelines sont un moyen très utile de voir l'avancement du projet, cela implique la répétition absurde de tests sur des parties qui n'ont pas besoin d'être retestées ou qui ont déjà été utilisés en local. Compiler le compilateur prend une moyenne de 1633413000 cycles. Lancer les tests du compilateur, donc lancer la commande *mvn test*, consomme autour de 18569430000 cycles. Ceci signifie 404 joules par test. Ainsi, dans notre projet, nous avons réalisé autour de 700 commits, ce qui aurait signifié 282 800 joules, soit un quart d'une batterie de voiture. C'est alors une grande dépense énergétique pour des tests qui ne sont souvent pas nécessaires, notamment pour des résolutions de bugs mineurs. Il est alors une bonne idée de ne pas utiliser cette option de validation.

Nous avons aussi pensé de réaliser des tests automatiques quotidiens, qu'à 7h soient lancés pour exporter un rapport de l'avancement des programmes dans chaque branche. Mais, pour la même raison que les Pipelines, nous avons estimé cette dépense comme étant excessive pour le peu d'utilité dans le projet : si nous avons en moyenne 7 branches parallèles dans le projet, le projet durant 20 jours entre le début et le jour du rendu, si on prend le même exemple de cycles de compilation (en moyenne, étant variable au long du projet), cette automatisation symboliserait une dépense de 56560 joules, ce qui correspond à 4 batteries de téléphone. Nous avons alors estimé que la dépense énergétique était trop élevée pour l'intérêt réel de l'automatisation, nous avons décidé de l'éviter.

2. Tests unitaires VS tests globaux

Lorsque nous réalisons des tests, nous devons étudier à l'avance ce que nous souhaitons tester à un moment précis. Ainsi, lorsque nous ajoutons un *print* dans une fonction de la partie A, ça ne sert absolument à rien de tester de manière spécifique les fonctions de la partie B et C. Non seulement cela signifie un ajout de temps dont nous n'avons pas besoin, mais de plus, cela signifie comme nous avons pu constater un surplus énergétique complètement abusif quant à l'objectif du test. Nous devons alors séparer les tests que nous voulions faire pour valider le code.

Comme nous avons vu, l'énergie consommée dépend du nombre de cycles utilisés par le processeur. Notre objectif est ainsi de réduire le nombre de cycles utilisés par le processeur. Pour cela, nous pouvions diviser nos tests en trois parties :


- Les tests unitaires : ce sont les tests qui valident une seule fonctionnalité du code.
- Les tests modulaires : ce sont les tests qui valident tout un module du code ; soit une partie en entier, soit une extension, soit plusieurs fonctionnalités combinées.
- Les tests globaux : ce sont des tests qui cherchent à valider l'entièreté du code développé.

Pour réaliser les tests modulaires et globaux, nous avons réalisé des scripts qui déclenchent les tests que nous souhaitons déclencher dépendant de ce que nous voulons valider.

Avec les scripts réalisés, nous avons suivi une politique de validation en plaçant le respect énergétique dans le centre de notre stratégie, en veillant à réduire le nombre de cycles du processeur :

- Toute solution de bug mineur entraîne uniquement le déclenchement d'un test unitaire. Un test plus important peut être réalisé si c'est réellement nécessaire, en choisissant de préférence les tests modulaires.
- Toute solution de bug majeur utilisera de préférence un test unitaire, mais ne doutera pas à utiliser un test modulaire, pouvant utiliser un test global si le bug affecte l'entièreté du projet.
- Après la fin du développement d'une fonctionnalité, si la fonctionnalité est indépendante il faut choisir un test unitaire. Sinon, des tests modulaires seraient la meilleure option.
- Finalement, lors des validations de la branche *develop* ou lors de la fin d'un module du projet, il faut passer par les tests globaux pour vérifier l'entièreté du code.

Ces règles implicites nous permettent de contrôler la dépense temporelle et énergétique lors de la réalisation des tests et de contrôler ainsi le nombre de cycles dépensés par notre processeur.



Nous avons aussi évité de recompiler notre code à chaque fois : nous exécutons le plus grand nombre possible de tests unitaires et modulaires avant la recompilation du code afin d'éviter des dépenses à nouveau pas nécessaires.

V. Extension

Notre objectif avec l'extension a été d'optimiser le plus possible le code exporté par le compilateur. Un code optimisé n'est pas seulement plus rapide, mais moins énergivore.

Dans un usage normal des compilateurs, un programme ne devrait être compilé qu'une seule fois pour développer un programme qui sera utilisé une ou plusieurs fois. Ainsi, un compilateur bien optimisé, mais résultant un programme pas optimisé n'est pas utile. Le plus important est de réduire la consommation dans l'exécution et la re-exécution du programme final.


Il est donc intéressant de considérer d'augmenter la dépense énergétique du compilateur afin de rendre un code final plus optimisé. Un tel code pourra être exécuté plusieurs fois en consommant beaucoup moins.

Ainsi, nos optimisations ont eu, comme un des objectifs (s'ils avaient fonctionné parfaitement comme ils ont été idées) rendre les programmes déterministes presque à coup zéro : les calculs auraient été faits en amont une seule fois, ce qui rendrait le code exporté extrêmement rapide. Par exemple, pour un programme comme *ln2.deca*, le code passerait de plus de 10000 cycles à 18 cycles (ce qui correspond à un simple print du résultat), car tous les calculs auraient été faits en amont. De même, un programme non déterministe réduirait considérablement le nombre de cycles. Certes, on n'atteindra jamais la rapidité d'un programme déterministe, mais le programme exporté sera moins énergivore ce qui peut signifier une vraie réduction en grande échelle.

À mode d'exemple, pour le code *syracuse.deca*, la version optimisée passe de 630797314 à 694840579 cycles, ce qui est une grande augmentation de la consommation du compilateur. Cependant, même si *syracuse.deca* est un programme déterministe, nous n'avons pas atteint les 18 cycles, mais il présente une diminution du nombre de cycles de plusieurs centaines de cycles.

Nous pouvons alors choisir le système de compilation selon l'objectif du programme :

Si nous souhaitons de compiler un programme qui sera souvent utilisé (par exemple, un programme commercial), l'option optimisée du compilateur est un bon choix, car même si la dépense initiale de compilation augmente considérablement, cette augmentation est justifiée par



la grande quantité de cycles qui ne seront pas utilisés après à chaque lancement du programme, ce qui symbolise beaucoup moins de dépenses énergétiques.

Néanmoins, si le programme ne sera utilisé qu'un nombre limité et faible de fois (par exemple, voir ponctuellement la valeur de $\ln 2$), la réduction de cycles n'est pas justifiée énergétiquement. Le surplus du compilateur est très supérieur au gain du programme optimisé. On va donc éviter l'option optimisée.

L'utilisateur pourra ainsi avoir le choix de compiler le programme selon son intérêt et ce qui impliquera le moindre de dépenses énergétiques possibles.

VI. Conclusion

Depuis le début du projet, nous avons voulu mettre l'aspect énergétique comme un enjeu majeur de notre objet. Pour cela, nous avons mis en œuvre plusieurs mesures pour réduire notre empreinte énergétique et faire un code le plus vert possible. Nous avons mis aussi en place un système de validation qui est aussi le plus respectueux possible de la planète. Pour la partie optimisation, le code optimisé provoque une augmentation de la consommation du compilateur. C'est à l'utilisateur de choisir si lancer le compilateur avec l'option d'optimisation ou pas selon ses intérêts.