ELEFTHERIOU Theophanis - ELIEZER Gwennan - GINDRIER Clément
PETITEAU Matthieu - POLICET Rémi

# ANALYSE DES IMPACTS ÉNERGÉTIQUES

Vous trouverez dans ce document les données relatives à la consommation énergétique de notre produit et les moyens mis en oeuvre pour réduire au maximum cette consommation

*Equipe GL60*

GRENOBLE INP UGA Ensimag

# SOMMAIRE

# Moyens mis en oeuvre pour évaluer la consommation énergétique

Afin d'évaluer notre consommation énergétique nous avons utilisé le calculateur de l'impact environnemental de l'INR. A l'aide de cet outil nous avons calculé notre impact environnemental sur la durée du projet (4 semaines). Il est de 81,32 kg eq CO2 ce qui fait un équivalent de 54,6 polaires forclaz ou une ampoule allumé pendant 178 jours. Cet impact est étonnamment bas, car la durée cumulé du projet est de 140 jours.

# Efficience du code produit

Afin de rendre notre compilateur aussi respectueux de l'environnement que possible dans le temps imparti, nous y avons réalisé plusieurs ajouts. Nous avons optimisé l'opérateur booléen unaire 'Not' avec la méthode de Morgan. Celle-ci consiste à inverser les 'And' en 'Or' au sein des champs d'action de l'opérateur 'Not'. Elle permet de gérer les 'Not' sans ajouter de cycles. Dans les cas où la méthode de Morgan ne peut être utilisé (appel de méthode par exemple), nous avons réalisé une méthode par défaut: celle-ci est moins optimale et réalise l'opération '1-<expression>'.

Au début du développement de notre compilateur, les opérations utilisaient seulement la pile pour dialoguer. Par exemple, toutes les expressions ajoutent un élément sur la pile, ainsi les autres opérations peuvent récupérer ce résultat en récupérant une valeur de la pile. Afin de rendre le code produit bien plus efficace que cela en termes de temps d'exécution et de coût énergétique, nous avons rapidement réalisé un outil de gestion de registres implémentant une pile virtuelle. Ce gestionnaire va supposer avoir les registres R2 à R16 (où à Rn avec l'option -r n) à sa disposition, ainsi que la pile, et va implémenter une pile virtuelle. Celle-ci permet, sans grandement modifier le code des programmes, d'utiliser les registres temporairement pour pour y sauvegarder des valeurs censées être dans la pile. Ainsi, quand une valeur est poussée sur la pile virtuelle, elle est en réalité déplacée vers un registre, et s'il n'y a plus de registre libre, alors le contenu du registre ayant l'élément ajouté à la pile virtuelle le plus tôt va déplacer cette valeur sur la vraie pile.
Cependant, seuls les registres R0 et R1 sont utilisables lorsque l'on utilise ce gestionnaire. On y a donc ajouté une fonctionnalité lui permettant de de donner un registre et de le récupérer plus tard. Il va donc donner un registre inutilisé s'il y en a un, ou le registre le plus "vieux" en stockant sa valeur sur la pile. Cette mécanique de don et récupération de registre peut être combinée avec la mécanique de pile virtuelle pour une meilleure optimisation. En effet, si un registre contient déjà la valeur à empiler et n'est pas R0 ou R1, elle peut être ajoutée à la pile virtuelle sans instruction supplémentaire. Il en va de même pour la récupération des éléments présents dans la pile virtuelle: si l'élément se trouve

physiquement sur un registre, récupérer ce registre depuis le gestionnaire ne génère aucune instruction.
Ce gestionnaire, implémenté dans la classe “RegisterManager”, nous a donc permis de gagner en performances sur de nombreuses opérations tout en gardant un code peu complexifié pour chaque opération.

# Efficience du procédé de fabrication

Afin de diminuer notre consommation énergétique au cours du projet, nous nous sommes concentrés sur nos processus. Nous avons cherché dès la fin de la première semaine du projet à optimiser nos tests tant que possible. Cette automatisation permet de limiter le temps passé à lancer des tests. Pour lancer tous nos tests il suffit d’utiliser la commande “tester” depuis un terminal placé dans le dossier Projet_GL. Nous avons ensuite classé nos tests dans 2 catégories: totalité des tests et tests de régression. Nous avons ajouté un script permettant de lancer nos tests par catégorie. Ainsi nous pouvions lancer seulement nos tests de régression pour vérifier qu’une modification n’avait abimé aucune des fonctionnalités de notre compilateur. Pour cela il suffit de lancer la commande “test_reg” depuis un terminal placé dans le dossier Projet_GL.
Notre deuxième mesure fut d’encadrer autant que possible notre usage informatique. Nous avons appliqué les différentes règles principales du Green IT: réduire le temps de mise en veille de nos PC personnels autant que possible, trouver un compromis sur la luminosité de nos machines (entre faible consommation d’énergie et protection des yeux de notre équipe), activer l’économisation de batterie sur chacune de nos machines personnelles, ne pas utiliser de deuxième écran quand cela n’est pas nécessaire.
Nous avons également fait un effort sur nos déplacements. Dès le début du projet nous nous sommes mis d’accord pour ne pas faire usage de la voiture et se restreindre tant que possible au tram, à la marche et au vélo. Tout au long du projet, ces normes ont été tenues par chacun des membres de l’équipe.

En fin de projet nous avons tous fait un calcul de bilan énergétique sur la durée totale de notre projet. Nous avons obtenu un résultat de:

# Analyse énergétique de l’extension

Durant l’analyse de notre extension, nous nous sommes posé la question de son impact énergétique. Notre première mesure fut de recréer une grammaire abstraite spécifique aux imports. Cette deuxième grammaire permet de ne parcourir que du code utile au cours de l’étape B. La grammaire concrète reste quant à elle inchangée. En effet, on souhaite tout de même pouvoir parcourir le contenu des fonctions et du programme principale dans l’étape

A. Cela permet de lire des fichiers avec un programme principal et des fonctions non vides d'être parcourus. Garder la grammaire concrète telle quelle est donc indispensable. En effet supprimer 'block' de cette grammaire et tout ce qui en découle uniquement ('list_inst', 'inst', 'if_then_else', etc…) entraînerait l'impossibilité d'importer des classes avec des fonctions non vides, ce qui est le but même de l'extension LINK.

Pour démontrer les avantages énergétique de notre deuxième grammaire, étudions les solutions pour implémenter l'extension sans celle-i. Si l'on avait utilisé qu'une seule grammaire abstraite, il aurait fallu parcourir en étape B le contenu de toutes les fonctions et du programme principal des fichiers importés. Cette méthode aurait certainement été plus longue à coder, à exécuter et à tester. Cette méthode nous a aussi semblé préférable car nos effectifs sur la partie A étaient plus élevés et que l'extension demandait plusieurs autres manipulations complexes en partie B. Ainsi la méthode avec deux grammaire est bien préférable car elle permet de gérer les imports simplement en diminuant le nombres de calculs.

En outre nous avons aussi envisagé la suppression des fichiers objets après la génération du fichier assembleur, lors de l'appel au linker. En effet, dans le cas ou l'on modifie l'un des fichiers deca de la chaîne d'imports, ils doivent tous être recompilé en fichier objet. Une fois que les fichiers objet ont été utilisés pour créer le fichier assembleur, ils ne sont plus d'aucune utilité. Cette deuxième optimisation permettrait donc de limiter la surcharge de la mémoire de la machine employant notre compilateur. Néanmoins le temps nous a manqué pour réaliser ce dernier ajout.