

Compilateur Deca — Analyse des impacts énergétiques

Projet Génie Logiciel – Équipe gl43

23 janvier 2026

Table des matières

1	Introduction	2
2	Méthodologie d'évaluation énergétique	2
2.1	Indicateurs retenus	2
2.2	Outils et protocole de mesure	2
3	Impact énergétique du code produit	2
3.1	Modèle énergétique retenu	2
3.2	Choix de génération de code	3
3.3	Impact de l'extension d'optimisation	3
4	Effizienz énergétique du processus de validation	3
4.1	Constat initial	3
4.2	Stratégie mise en œuvre	3
4.3	Compromis et garanties	4
5	Impact de l'extension d'optimisation sur le code IMA généré	4
5.1	Rôle de l'optimisation dans la réduction de l'impact énergétique	4
5.2	Limites de l'optimisation et pistes d'amélioration	4
6	Conclusion	5

1 Introduction

Le projet Génie Logiciel consiste à concevoir et implémenter un compilateur pour le langage Deca.

Si la priorité du projet a d’abord été d’obtenir un compilateur correct et fonctionnel, nous avons progressivement intégré une réflexion sur l’impact énergétique de notre travail. Cette réflexion concerne à la fois les programmes générés par le compilateur et le processus de développement et de validation du compilateur lui-même.

Nous avons ainsi cherché à limiter les calculs inutiles et à faire des choix techniques plus sobres, sans compromettre la qualité ni la fiabilité du compilateur. Ces choix se traduisent notamment par une extension d’optimisation du code généré, ainsi que par une organisation raisonnée des campagnes de tests.

Ce document présente l’analyse de ces impacts énergétiques et les mesures mises en œuvre dans le cadre du projet.

2 Méthodologie d’évaluation énergétique

L’analyse énergétique menée dans le cadre du projet repose sur des indicateurs indirects, adaptés au contexte du compilateur `decac` et aux outils disponibles.

2.1 Indicateurs retenus

Pour le code généré, le processeur virtuel IMA fournit un modèle d’exécution basé sur un nombre de cycles par instruction. Le nombre total de cycles exécutés est utilisé comme une approximation de la consommation énergétique d’un programme.

Pour le processus de développement et de validation, des informations indirectes sur la consommation de ressources sont obtenues à l’aide de la commande `/usr/bin/time -v`. Les indicateurs exploités incluent notamment le temps CPU utilisateur et système, le temps d’exécution réel, ainsi que des données relatives à l’activité mémoire.

2.2 Outils et protocole de mesure

Les mesures sont réalisées lors de l’exécution des campagnes de tests globales, en particulier lors du lancement du script principal de validation `test_all.sh`. Les résultats fournis par `/usr/bin/time` sont utilisés à des fins comparatives, afin de caractériser le coût des campagnes complètes et de justifier la mise en place de stratégies visant à réduire les exécutions inutiles.

3 Impact énergétique du code produit

Cette section analyse l’impact énergétique des choix de compilation du langage Deca vers le processeur virtuel IMA.

3.1 Modèle énergétique retenu

Le processeur virtuel IMA associe à chaque instruction un coût en nombre de cycles. Ce modèle permet d’identifier les instructions et les séquences les plus coûteuses, et d’évaluer l’impact des transformations opérées par le compilateur. Dans ce cadre, réduire le nombre total d’instructions exécutées, en particulier celles à coût élevé, constitue un levier direct de réduction de la consommation énergétique.

3.2 Choix de génération de code

Les choix de génération de code privilégient des séquences simples et robustes, tout en limitant autant que possible les instructions inutiles. Une attention particulière est portée à la gestion des registres et de la pile, afin d'éviter des accès mémoire superflus et des opérations de sauvegarde ou de restauration redondantes.

Ces choix contribuent à limiter le nombre de cycles exécutés et donc le coût énergétique des programmes générés, même en cas de non activation de l'option `-o` d'optimisation.

3.3 Impact de l'extension d'optimisation

L'extension développée dans le cadre du projet vise à optimiser le code généré en supprimant des instructions sans effet sur la sémantique du programme. Elle permet notamment :

- l'élimination de code mort ;
- la suppression de séquences `LOAD/STORE` inutiles ;
- la réduction globale du nombre d'instructions exécutées (simplifications algébriques, propagation de variables et de constantes, constant folding).

Ces transformations réduisent directement le nombre de cycles IMA exécutés lors de l'exécution d'un programme, et ont donc un impact énergétique positif. L'extension agit ainsi comme un levier simple et efficace de sobriété énergétique, sans compromettre la correction du code généré.

4 Efficience énergétique du processus de validation

Le processus de développement et de validation du compilateur constitue une source significative de consommation de ressources. Dans le cadre du projet, les campagnes de tests impliquent de nombreuses compilations et exécutions, susceptibles d'entraîner des calculs redondants.

4.1 Constat initial

Les campagnes de validation complètes, incluant la compilation du compilateur, l'exécution des tests et la génération des rapports, mobilisent fortement le processeur et le système. Les mesures indirectes réalisées à l'aide de la commande `/usr/bin/time` mettent en évidence un temps CPU important, ainsi qu'une activité significative liée aux accès mémoire.

Ce constat a motivé une réflexion sur l'organisation des tests, afin de limiter les exécutions inutiles sans réduire le niveau de confiance dans la validation.

4.2 Stratégie mise en œuvre

Pour réduire l'impact énergétique du processus de validation, des scripts de tests à relance conditionnelle ont été mis en place. Leur principe repose sur l'analyse des fichiers modifiés depuis la dernière validation, afin de déterminer précisément quelles campagnes de tests doivent être relancées.

Les stratégies adoptées incluent notamment :

- la distinction entre modifications fonctionnelles et changements sans impact (commentaires, mise en forme) ;
- le ciblage des tests en fonction des sous-systèmes affectés (lexical, syntaxique, contextuel, génération de code) ;
- la non-relance des campagnes complètes en l'absence de changements pertinents.

Par exemple, si un test `.deca` ou le `.expected` correspondant a été effectivement modifié, seul ce test est relancé. De même, si une modification fonctionnelle a été faite au niveau de `fr.ensimag.deca.codegen`, seuls les tests de génération de code sont relancés.

Les modifications en question sont déterminées en comparaison avec non seulement les changements locaux, mais aussi le dernier commit sur Git.

4.3 Compromis et garanties

Cette approche permet de réduire significativement le nombre d'exécutions de tests, et donc la consommation de ressources associée, tout en conservant un effort de validation adapté aux modifications apportées. En cas de changement critique ou avant un rendu, des campagnes complètes sont systématiquement relancées afin de garantir la qualité globale du compilateur.

5 Impact de l'extension d'optimisation sur le code IMA généré

5.1 Rôle de l'optimisation dans la réduction de l'impact énergétique

L'extension développée dans le cadre de ce projet vise à améliorer l'efficacité du code IMA produit par le compilateur **decac**, en supprimant des instructions inutiles ou redondantes générées lors de la phase de génération de code. Cette démarche s'inscrit directement dans une logique de réduction de la consommation énergétique des programmes compilés, la consommation étant ici approximée par le nombre de cycles exécutés sur le processeur virtuel IMA.

Concrètement, l'optimisation repose principalement sur :

- l'élimination de code mort, c'est-à-dire des instructions dont les résultats ne sont jamais utilisés ;
- la suppression de séquences redondantes de type **LOAD/STORE**, lorsque la valeur chargée est déjà disponible dans un registre et n'a pas été modifiée ;
- la simplification de certaines suites d'instructions arithmétiques ou de contrôle n'ayant aucun effet observable sur l'état du programme.

Ces transformations ont un impact direct sur le nombre total d'instructions exécutées. En particulier, la réduction du nombre d'accès mémoire (**LOAD, STORE**) est significative d'un point de vue énergétique. De plus, la suppression d'instructions inutiles permet d'éviter des cycles processeur superflus lors de l'exécution.

Il est à noter que certains mécanismes d'optimisation ne sont pertinents que pour des programmes Deca sans objet. C'est pourquoi une vérification préalable est effectuée pour déterminer la nature du programme Deca (avec ou sans objet) afin de n'appliquer les optimisations que lorsque c'est judicieux, réduisant ainsi des coûts inutiles à la compilation.

En s'appuyant sur les coûts en cycles fournis pour l'architecture IMA, on peut raisonnablement conclure que la diminution du nombre d'instructions générées, combinée à une meilleure sélection de celles-ci, entraîne une baisse du nombre total de cycles exécutés. L'extension d'optimisation contribue ainsi à produire un code IMA plus compact, plus efficace et moins énergivore, sans compromettre la correction fonctionnelle des programmes compilés.

On note cependant que nous faisons là un compromis avec la compilation ; une hausse du coût de la compilation qui contraste avec une baisse du coût de l'exécution. Ce compromis est en l'occurrence assumé, dans la mesure où l'on considère que l'exécution se répèterait plusieurs fois, contrairement à la compilation qui s'effectuerait, normalement, une seule fois.

5.2 Limites de l'optimisation et pistes d'amélioration

Bien que l'extension développée permet de réduire significativement le nombre d'instructions inutiles et redondantes, certaines optimisations plus fines n'ont pas été implémentées dans le cadre de ce projet.

En particulier, nous n'avons pas mis en œuvre d'optimisations arithmétiques sémantiques fondées sur le modèle de coût de l'architecture IMA. Par exemple, des opérations telles que la multiplication, la division ou le modulo par des puissances de 2 auraient pu être remplacées par

des instructions de décalage (**SHL**, **SHR**), dont le coût en cycles est nettement inférieur à celui des instructions **MUL**, **QUO** ou **REM**. Une telle transformation aurait permis de réduire davantage le nombre de cycles exécutés, et donc la consommation énergétique estimée des programmes générés, avec cependant un compromis avec la compilation.

Ces optimisations nécessitent toutefois une analyse plus fine des expressions (gestion des signes, des types, des cas limites) et une propagation de constantes plus avancée, ce qui dépassait le périmètre raisonnable du projet au regard des contraintes de temps et de complexité. Elles constituent néanmoins des perspectives d'amélioration pertinentes pour prolonger ce travail.

6 Conclusion

Durant ce projet, nous avons non seulement développé des compétences techniques liées à la conception d'un logiciel complexe tel qu'un compilateur, mais également pris conscience de l'impact énergétique de nos choix de conception et de validation. Nous avons ainsi mis en œuvre différentes stratégies, certes perfectibles, visant à en limiter les effets et à produire un compilateur plus respectueux des enjeux énergétiques.