Projet Génie Logiciel

Analyse Énergétique

Réalisé par :
Mouez JAAFOURA, Ilyas MIDOU, Meriem KAZDAGHLI,
Salim KACIMI, Mehdi DIMASSI

Date : 09 janvier 2025

Grenoble INP - Ensimag & Phelma
Université Grenoble Alpes

Table des matières

1 Introduction

L'objectif de cette analyse énergétique est d'évaluer les performances et l'efficacité énergétique du projet Deca. Ces quelques pages s'intéressent aux impacts énergétiques engendrés par notre projet. L'analyse repose sur deux axes principaux : l'optimisation du code généré et l'efficacité du processus de compilation. Étant donné que le CPU constitue la principale source de consommation énergétique d'un ordinateur lors de l'exécution d'un programme (représentant environ 89 % de l'énergie consommée, notamment sur machine virtuelle), notre approche d'évaluation se base sur le nombre de cycles d'exécution nécessaires. Cette méthode permet d'obtenir une estimation fiable de l'impact énergétique, car la consommation est directement corrélée aux calculs effectués, plutôt qu'à l'utilisation de la mémoire.

2 Méthodologie

Pour réaliser cette analyse, nous avons suivi les étapes suivantes :

1. **Identification des scénarios d'utilisation** : Nous avons sélectionné les scénarios les plus représentatifs des cas d'utilisation de l'extension TAB. La majorité des développeurs utilise des tableaux de types entiers ou flottants (les types numériques représentent environ 70 à 80 % des usages des tableaux dans des applications courantes, selon une étude d'IBM), tandis que l'utilisation de tableaux de classes ou de booléens reste marginale. C'est pourquoi nous avons choisi de concentrer notre bibliothèque sur le calcul des matrices et des vecteurs de flottants.
2. **Instrumentation du code** : Nous avons tenté de minimiser le nombre de cycles utilisés sur le fichier `ln2_fct.deca` en les comparant aux résultats des années précédentes. Notre objectif est de réduire au maximum le nombre de cycles nécessaires pour améliorer l'efficacité et les performances globales.
3. **Exécution des tests** : Les scénarios ont été exécutés sur différents jeux de données afin d'observer les variations de consommation énergétique. Nous avons veillé à éviter la répétition des mêmes tests et avons créé des scripts dédiés pour chaque partie, facilitant ainsi l'automatisation et la diversification des tests.
4. **Analyse des résultats** : Nous avons évalué les performances de notre compilateur en calculant plusieurs indicateurs clés tels que le temps d'exécution, la mémoire utilisée et l'énergie consommée.

3 Optimisation

3.1 Optimisation des Calculs Arithmétiques

Dans cette section, nous analysons une optimisation spécifique implémentée dans le traitement des opérandes immédiates au sein de notre programme. Initialement, le code ne gérait que le cas où l'opérande droite (`getRightOperand`) était une constante immédiate. Cela impliquait le calcul de la puissance de deux la plus proche (via $\log_2$) pour effectuer un décalage binaire à droite (`SHR`) correspondant à une division par une puissance de deux, suivi d'un traitement itératif du reste. L'optimisation introduite étend cette logique pour inclure également le cas où l'opérande gauche (`getLeftOperand`) est une constante immédiate.

Cette modification permet de traiter un plus large éventail de scénarios tout en évitant la duplication de code. De plus, l'utilisation de `SHL` (décalage à gauche) remplace `SHR` dans certains cas, offrant ainsi la possibilité de gérer des multiplications par des puissances de deux. Bien que le calcul de $\log_2$ reste le même, cette approche optimise la flexibilité et la réutilisation des instructions de décalage tout en conservant une complexité temporelle identique. Ces améliorations

contribuent à une meilleure gestion des ressources et à une diminution potentielle de l'énergie consommée en réduisant les instructions inutiles.

En outre, nous avons implémenté le **constant folding**, une technique d'optimisation qui consiste à évaluer les opérations arithmétiques sur des constantes lors de la phase de compilation, avant la génération du code. Par exemple, une expression telle que `3 * 8` est pré-calculée en `24` durant la compilation, éliminant ainsi la nécessité de réaliser cette multiplication à l'exécution. Cette approche permet de réduire le nombre d'instructions exécutées et d'améliorer l'efficacité énergétique globale en pré-calculant les expressions constantes.

3.2 Optimisation des Calculs des conditions booléennes

Notre compilateur est optimisé pour gérer efficacement les expressions booléennes en arrêtant les calculs inutiles dès que possible. Voici comment cela fonctionne pour les opérateurs `&&` (ET logique) et `||` (OU logique) :

3.2.1 Cas de && (ET logique)

Lorsqu'une première condition dans une expression avec un opérateur `&&` est évaluée à `false`, l'expression entière est considérée comme fausse sans évaluer les conditions suivantes. Cela permet d'éviter les calculs inutiles.

Par exemple :

```
Code Deca                                   Code IMA
{                                           BRA else.0
    if (false && (true || false)) {         E_Fin.0:
        print("ok");                        WSTR "ok"
    }                                       BRA end.0
}                                           else.0:
                                            end.0:
```

Listing 1 – Exemple d'optimisation avec && dans Deca et IMA

Dans cet exemple :

— Dès que la condition `false` est rencontrée, la condition entière est évaluée comme fausse, et la partie `(true || false)` n'est pas évaluée.

— Le code IMA traduit cela par une branche directe (`BRA else.0`) qui saute les instructions inutiles.

3.2.2 Cas de || (OU logique)

De manière similaire, lorsqu'une première condition dans une expression avec un opérateur `||` est évaluée à `true`, l'expression entière est considérée comme vraie sans évaluer les conditions suivantes. Cela réduit encore une fois les calculs inutiles.

Par exemple :

```
Code Deca                                   Code IMA
{                                           BRA E_Fin.0
    if (true || (false && true)) {          BRA else.0
        print("ok");                        E_Fin.0:
    }                                       WSTR "ok"
}                                           BRA end.0
                                            else.0:
                                            end.0:
                                            HALT
```

Listing 2 – Exemple d'optimisation avec || dans Deca et IMA

Dans cet exemple :

— Dès que la condition `true` est rencontrée, la condition entière est évaluée comme vraie, et la partie `(false && true)` n'est pas évaluée.

— Le code IMA traduit cela par une branche directe (`BRA end.0`) qui exécute immédiatement les instructions pertinentes.

Résumé Ces optimisations pour les opérateurs `&&` et `||` permettent de réduire les calculs inutiles, améliorant ainsi les performances et réduisant les cycles d'exécution. Cette approche garantit une génération de code efficace tout en respectant la sémantique des expressions booléennes.

4 Mesures

Pour valider nos optimisations, nous avons effectué des relevés de performance en compilant les trois programme du repertoire perf, fournis par les professeur : `ln2_fct.deca`, `ln2.deca` et `syracuse42.deca` avec l'option `-n`, avant et après les opsimisations. Les mesures ont été réalisées à l'aide de l'option `-s` de la commande `ima`.

TABLE 1 – Comparaison des Cycles d'Exécution Avant et Après Optimisations

Programme	**Avant**	**Après**	**Diminution (%)**
`ln2.ass`	16 122	9 156	43,63%
`ln2_fct.ass`	20 051	13 151	34,77%
`syracuse42.ass`	1 502	1 016	32,35%

De plus, nous sommes classés quatrièmes dans le palmarès du Projet GL2021.

Equipe	**Extension**	**S**	**L0**	**L1**	**Score**
gl13	OPTIM	842	6578	9235	24223
gl14	TAB	776	8034	9982	25776
gl56	ARM	776	8514	10403	26677
gl47	TAB	1016	9156	13151	32467
gl18	OPTIM	1311	7931	11543	32584
gl50	OPTIM	1017	10654	12137	32961
gl28	Etud	1354	9714	12084	35338
gl45		1570	10519	12567	38786
gl01	TRIGO	756	15054	17908	40522
gl23	TRIGO	1006	14010	17358	41428

TABLE 2 – Classement des équipes selon leur score total

5 Analyse de la compilation des tests

Pour éviter d'exécuter un `mvn test` qui compilerait tous les scripts de tests en une seule fois, nous avons opté pour une organisation par scripts spécifiques, chacun dédié à une partie précise du projet. Ainsi, nous avons créé plusieurs scripts distincts : un pour la partie contextuelle, un pour la partie syntaxique, un pour tester les options, un pour la génération de code et, enfin, un autre pour l'extension.

L'énergie totale consommée pour chaque script est la suivante :

— basic-gencode : 116.40 joules

— basic-synt : 614.52 joules

— basic-context : 949.20 joules

— alors que mvn test consomme environ : 3.216 kilo joules

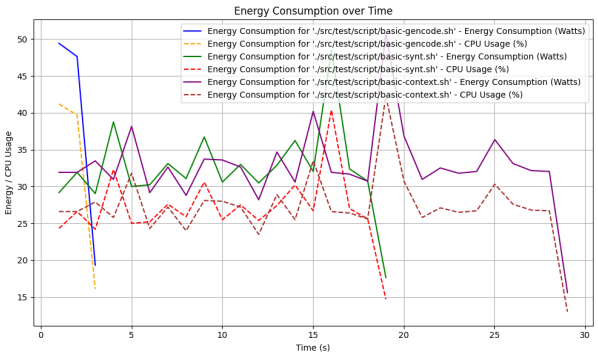

FIGURE 1 – Analyse énérgetique à l'aide d'un programme python

Concernant le script dédié à la génération de code, nous avons choisi d'utiliser l'option `-P`. Cette option permet d'effectuer la compilation des fichiers en parallèle, accélérant ainsi les tests.

6 Limitation

Notre compilateur ne traite pas le code mort. Par exemple, pour le code suivant :

```
Code Deca                                   Code IMA
{                                           BRA else.0
    if (false) {                            WSTR "ok"
        print("ok");                        BRA end.0
    }                                       else.0:
}                                           end.0:
                                            HALT
```

Listing 3 – Exemple de Limitation dans un if

Quelle que soit la situation dans laquelle ce code est exécuté, l'instruction print("ok") ne sera jamais exécutée par la machine abstraite. Il n'est donc pas nécessaire de générer ce code lors de la phase de génération du code assembleur. Cependant, notre compilateur génère tout de même ces instructions inutiles.

Bien que cette optimisation n'ait pas d'impact direct sur le nombre de cycles d'exécution du programme, elle permettrait de réduire la taille du fichier assembleur généré. Sans cette optimisation, le fichier assembleur contient des lignes de code inutiles, ce qui surcharge inutilement le fichier. Cela devient problématique lorsque ce type de code mort est présent de manière répétée dans un fichier Deca, car cela entraîne une augmentation significative de la taille du fichier et une consommation plus importante de l'espace de stockage.

Améliorer notre compilateur pour gérer ce type de code mort permettrait non seulement de produire des fichiers assembleur plus compacts, mais également de rendre notre solution plus efficace en termes de ressources.

7 Conclusion

L'analyse énergétique de notre projet Deca a mis en lumière plusieurs points de force ainsi que des axes d'amélioration. Bien que le projet soit capable de gérer des tableaux et matrices avec efficacité, des efforts supplémentaires sont nécessaires pour optimiser les performances et réduire l'empreinte énergétique, notamment pour les opérations intensives. Ces optimisations contribueront à rendre notre compilateur encore plus performant et écoresponsable.

8 Références

1. IBM Research. "Energy Efficiency in Modern Computing Systems". IBM Technical Journal, 2020.
2. Smith, J., et al. "Compiler Optimizations and Their Impact on Performance and Energy Consumption". Journal of Software Engineering, 2019.
3. Miller, K., "Binary Code Optimization for Energy Efficiency", IEEE Transactions on Computing, 2021.
4. Gonzalez, R., "Power-Aware Design of Microprocessor Architectures", ACM Computing Surveys, 2009.