ENSIMAG

Projet Génie Logiciel

Groupe 13

Analyse Énergétique et Optimisation

Réalisé par :

Benjelloun Otman
Birée Thomas
Boulouz Amine
El Goumiri Rida
Loginova Aleksandra

Année Universitaire : 2024/2025

Table des matières

1 Introduction **2**

2 Méthodologie d'évaluation **2**
2.1 Outils de mesure 2
2.2 Stratégies d'optimisation 2
2.3 Processus de validation et réduction de l'impact énergétique 3

3 Analyse énergétique des programmes Deca **4**
3.1 Programme `ln2` 4
3.2 Programme `ln2_fct` 4
3.3 Programme `syracuse42` 4
3.4 Analyse avec `perf stat` 5

4 Analyse de l'impact du nombre de registres sur la consommation énergétique **6**
4.1 Résultats des tests 6
4.2 Analyse des résultats 6
4.3 Conclusion 6

5 Tests de performance avec `time` **7**
5.1 Test 1 : Compilation avec l'option parallèle `-P` 7
5.2 Test 2 : Compilation standard du programme `ln2` 7
5.3 Test 3 : Compilation standard du programme `ln2_fct` 7
5.4 Analyse comparative 7
5.5 Conclusion sur les tests avec `time` 8
5.6 Discussion sur l'efficience énergétique 8

6 Optimisation énergétique avec ARM **8**

7 Recommandations pour améliorer l'efficience énergétique **8**

1 Introduction

Dans un contexte où les préoccupations environnementales deviennent cruciales, l'efficacité énergétique des logiciels constitue un enjeu majeur du développement informatique. Notre projet de compilateur Deca, bien qu'étant avant tout un exercice pédagogique, offre une opportunité d'explorer et d'analyser l'impact énergétique des choix de conception et d'implémentation en génie logiciel.

Cette analyse se concentre sur trois axes principaux : l'efficience du code produit par notre compilateur, l'optimisation de notre processus de développement et de validation, et l'impact énergétique de notre extension ARM. Notre objectif est double : évaluer quantitativement la consommation énergétique de notre solution et identifier les stratégies d'optimisation les plus pertinentes.

2 Méthodologie d'évaluation

2.1 Outils de mesure

Pour évaluer la consommation énergétique de notre compilateur, nous utilisons plusieurs outils principaux :

- `perf stat` : Cet outil Linux mesure divers indicateurs de performance du processeur, notamment :
  - Les cycles CPU totaux.
  - Les instructions par cycle (IPC), indiquant l'efficacité du processeur.
  - Les défauts de cache (*stalled cycles* sur le frontend ou backend).
  - Les branchements et erreurs de prédiction (*branch-misses*).
- `ima -s` : Cette commande fournit :
  - Le nombre d'instructions exécutées.
  - Le temps d'exécution simulé en cycles.
- `time` : Permet de mesurer le temps d'exécution global (`real`) et la répartition entre CPU utilisateur (`user`) et système (`sys`).
- `-P` : L'option parallèle de `decac`, utilisée pour analyser l'impact de la compilation simultanée de plusieurs fichiers source.
- `-r` : L'option permettant de spécifier le nombre de registres alloués (`-r 5`, `-r 8`, `-r 16`) pour évaluer son effet sur les instructions générées, les cycles consommés et la consommation énergétique.

2.2 Stratégies d'optimisation

L'optimisation du compilateur repose sur deux axes principaux : améliorer la qualité du code généré et optimiser l'utilisation des ressources matérielles.

Optimisations de code

Pour garantir un code efficace et maintenable, nous avons adopté les stratégies suivantes :

— **Architecture orientée objet** : Utilisation des principes d'héritage en Java pour limiter la redondance et optimiser la mémoire.

— **Modularité** : Division du code en composants indépendants, facilitant sa maintenance et son évolutivité.

— **Optimisations ciblées** : Amélioration des sections critiques pour maximiser leur impact sur les performances.

Gestion des registres

Une allocation efficace des registres est essentielle pour limiter les cycles inutiles et réduire la consommation énergétique :

— Réduction des accès mémoire en stockant un maximum de données dans les registres.

— Minimisation des transferts entre registres et mémoire pour réduire les instructions superflues.

Ces stratégies permettent d'optimiser à la fois les performances et l'efficacité énergétique des programmes générés par le compilateur.

2.3 Processus de validation et réduction de l'impact énergétique

Lors du développement de notre compilateur, nous avons soigneusement conçu notre processus de validation pour minimiser l'impact énergétique tout en garantissant une qualité optimale.

Bien que l'intégration d'un pipeline automatisé sur Git, déclenchant une batterie de tests à chaque commit, aurait permis de tester en continu les modifications, nous avons choisi de ne pas le mettre en place. Ce choix s'inscrit dans une démarche éco-responsable, visant à éviter des tests superflus, en particulier lors de commits mineurs impliquant peu de changements.

À la place, notre approche a été la suivante :

— **Exécution manuelle des tests** : Nous lançons les scripts de validation manuellement pour parcourir l'ensemble des tests. Ce processus est réalisé une seule fois avant des changements majeurs, ou avant un `push` vers le dépôt Git, afin de limiter les ressources utilisées.

— **Validation ciblée des améliorations** : Pour des tests spécifiques ou des validations ponctuelles, nous ne lançons qu'un seul test à la fois, réduisant ainsi la consommation énergétique.

— **Tests complets avant intégration** : Avant chaque mise à jour significative sur le dépôt, nous exécutons une validation complète pour nous assurer de la stabilité et de la fiabilité du compilateur.

Cette méthodologie garantit un équilibre entre une consommation énergétique réduite et un effort de validation rigoureux, contribuant à la qualité du compilateur sans générer de surcoût énergétique inutile. En intégrant cette approche manuelle et réfléchie, nous répondons à l'exigence croissante d'un développement logiciel éco-responsable.

3 Analyse énergétique des programmes Deca

Dans cette section, nous analysons trois programmes représentatifs exécutés sur la machine virtuelle IMA pour évaluer l'impact des choix de conception, des optimisations et de la complexité algorithmique sur la consommation énergétique.

3.1 Programme `ln2`

Le programme `ln2` calcule une approximation de ln(2) par dichotomie en utilisant le développement en série de la fonction exponentielle jusqu'à l'ordre 7.

```
6.93148e-01 = 0x1.62e448p-1
Nombre d'instructions :      123
Temps d'exécution :       19404 cycles
```

Ce programme exécute 123 instructions avec un temps simulé de 19404 cycles. Sa faible consommation en instructions est due à la simplicité relative du calcul. Cependant, l'utilisation répétée d'opérations arithmétiques complexes (multiplications et divisions flottantes) augmente la consommation en cycles.

3.2 Programme `ln2_fct`

Le programme `ln2_fct` est une version refactorisée de `ln2`, utilisant une classe et une méthode pour encapsuler le calcul du polynôme exponentiel.

```
6.93148e-01 = 0x1.62e448p-1
Nombre d'instructions :      209
Temps d'exécution :       23071 cycles
```

La refactorisation introduit une abstraction supplémentaire via l'objet `Polyexp`, augmentant ainsi le nombre d'instructions exécutées à 209 (+69,92%). Le temps d'exécution simulé augmente également, atteignant 23071 cycles.

Le surcoût énergétique est dû principalement aux appels de méthode et à la gestion des objets, caractéristiques des abstractions orientées objet.

Comparaison `ln2` vs `ln2_fct`

La version fonctionnelle (`ln2_fct`) présente un coût énergétique plus élevé, illustrant l'impact des abstractions orientées objet sur la performance. Cependant, elle offre une meilleure lisibilité et modularité, ce qui facilite la maintenance et l'évolution du code.

3.3 Programme `syracuse42`

Le programme `syracuse42` calcule le temps de la suite de Syracuse pour la valeur initiale 42.

```
Résultat : 8
Nombre d'instructions :       58
Temps d'exécution :         1751 cycles
```

Avec seulement 58 instructions et un temps d'exécution de 1751 cycles, `syracuse42` est le programme le plus léger de cette analyse. Ces résultats ont été obtenus avec la commande suivante :

Cela s'explique par sa nature itérative simple, sans utilisation de calculs flottants coûteux.

Comparaison globale

- **Nombre d'instructions** :
  - `syracuse42` : 58
  - `ln2` : 123
  - `ln2_fct` : 209
- **Temps d'exécution simulé (en cycles)** :
  - `syracuse42` : 1751
  - `ln2` : 19404
  - `ln2_fct` : 23071

3.4 Analyse avec `perf stat`

Pour compléter notre évaluation énergétique, nous avons utilisé l'outil `perf stat` afin d'obtenir des indicateurs précis sur la performance des programmes Deca. Cette méthode nous a permis d'analyser en détail les cycles processeur, les instructions exécutées et les défauts de cache pour évaluer l'efficience des programmes générés par notre compilateur.

Programme `ln2`

- **Cycles CPU** : 24,799,788
- **Instructions exécutées** : 23,332,090
- **IPC (instructions par cycle)** : 0.94
- **Branch-misses** : 8.30%

Programme `ln2_fct`

- **Cycles CPU** : 28,161,997
- **Instructions exécutées** : 28,111,503
- **IPC (instructions par cycle)** : 1.00
- **Branch-misses** : 7.51%

Programme `syracuse42`

- **Cycles CPU** : 24,285,640
- **Instructions exécutées** : 13,858,965
- **IPC (instructions par cycle)** : 0.57
- **Branch-misses** : 9.89%

Comparaison avec `ima -s`

Les résultats obtenus avec `perf stat` sont cohérents avec ceux de `ima -s` et offrent des informations pour comprendre les impacts énergétiques et structurels des programmes compilés.

4 Analyse de l'impact du nombre de registres sur la consommation énergétique

Cette section analyse les résultats obtenus avec l'option `-r` du compilateur `decac`, qui permet de spécifier le nombre de registres alloués. Les tests sur les programmes `ln2` et `syracuse42` illustrent l'effet du nombre de registres sur les instructions générées, le temps d'exécution et la consommation énergétique.

4.1 Résultats des tests

- `ln2` **avec** `-r 5` : Instructions : 211, Temps : 25402 cycles.
- `ln2` **avec** `-r 8` : Instructions : 177, Temps : 23242 cycles.
- `syracuse42` **avec** `-r 16` : Instructions : 58, Temps : 1751 cycles.

4.2 Analyse des résultats

Impact sur `ln2`

Avec `-r 5`, le programme génère 211 instructions pour 25402 cycles. Augmenter les registres à `-r 8` réduit les instructions à 177 et le temps à 23242 cycles. Cela s'explique par :

- Une réduction des accès mémoire grâce à des registres supplémentaires pour stocker les variables temporaires.
- Une diminution des cycles de latence liés aux opérations mémoire.

Cependant, les calculs complexes nécessaires au développement en série rendent `ln2` énergétiquement coûteux, même avec des optimisations.

Impact sur `syracuse42`

Avec `-r 16`, le programme génère 58 instructions et s'exécute en 1751 cycles. Sa simplicité algorithmique (boucles conditionnelles, absence de calculs flottants) permet une utilisation optimale des registres, minimisant les accès mémoire et le nombre d'instructions.

4.3 Conclusion

L'augmentation des registres réduit le nombre d'instructions et le temps d'exécution grâce à une meilleure utilisation des ressources processeur et moins de sauvegardes en mémoire.

- **Programmes simples** (`syracuse42`) : Maximisent l'efficacité énergétique en exploitant pleinement les registres.

— **Programmes complexes** (`ln2`) : Bien qu'améliorés, leur coût en cycles reste élevé à cause des calculs intensifs.

Ces résultats soulignent l'importance d'ajuster le nombre de registres en fonction de la complexité des programmes pour équilibrer performance et consommation énergétique.

5 Tests de performance avec `time`

Pour compléter notre analyse énergétique, nous avons utilisé l'outil `time` pour mesurer les temps d'exécution de notre compilateur Deca et des programmes générés. Les résultats des tests sont présentés ci-dessous :

5.1 Test 1 : Compilation avec l'option parallèle `-P`

```
real    0m0.395s
user    0m0.739s
sys     0m0.101s
```

Ce test montre un gain de performance grâce à l'utilisation de l'option `-P`, qui permet de paralléliser la compilation de plusieurs fichiers. Bien que le temps utilisateur (`user`) augmente légèrement, le temps réel (`real`) reste compétitif, illustrant une bonne répartition des tâches.

5.2 Test 2 : Compilation standard du programme `ln2`

```
real    0m0.359s
user    0m0.645s
sys     0m0.083s
```

Ce test illustre les performances de la compilation standard d'un seul fichier. Le temps utilisateur est inférieur à celui de l'option parallèle, mais le temps réel reste comparable. Cela indique que la parallélisation est particulièrement avantageuse pour plusieurs fichiers.

5.3 Test 3 : Compilation standard du programme `ln2_fct`

```
real    0m0.351s
user    0m0.675s
sys     0m0.095s
```

Ce test révèle une performance similaire à celle du programme `ln2`, avec des temps légèrement inférieurs. Cela reflète l'impact modéré de l'abstraction supplémentaire introduite par la classe `Polyexp`.

5.4 Analyse comparative

Les résultats montrent que l'option `-P` améliore significativement les performances lorsqu'il s'agit de compiler plusieurs fichiers simultanément. Cependant, pour un fichier unique, l'impact de la parallélisation est limité. Ces données soulignent l'intérêt d'utiliser des options d'optimisation adaptées en fonction de la taille et du type de projet.

5.5 Conclusion sur les tests avec `time`

Ces mesures illustrent l'importance des optimisations parallèles et des temps de traitement différenciés entre les tâches (analyse, compilation et exécution). Pour des fichiers de grande taille ou en lots, l'utilisation de `-P` est recommandée pour ameliorer les performances.

5.6 Discussion sur l'efficience énergétique

Les résultats montrent que :

— La gestion des objets (comme dans `ln2_fct`) augmente significativement la consommation énergétique.

— Les calculs arithmétiques complexes, en particulier les flottants, contribuent de manière importante au coût en cycles.

— Les boucles avec des instructions simples (comme dans `syracuse42`) ont un impact énergétique minime.

Ces observations soulignent l'importance de choisir des optimisations adaptées au contexte d'exécution : les programmes embarqués ou contraints en énergie bénéficieraient d'une approche plus minimaliste comme `syracuse42`.

6 Optimisation énergétique avec ARM

L'architecture ARM, avec son jeu d'instructions RISC (Reduced Instruction Set Computing), offre des avantages significatifs pour l'optimisation énergétique de notre compilateur. La simplicité et l'uniformité des instructions ARM permettent une meilleure prédiction des branches et une utilisation plus efficace du pipeline, réduisant ainsi la consommation d'énergie. Les instructions de charge-stockage optimisées et la gestion fine des registres contribuent également à minimiser les accès mémoire, source majeure de consommation énergétique. De plus, les modes d'exécution conditionnelle d'ARM réduisent les branchements, améliorant l'efficacité du code généré. Cette architecture, largement adoptée dans l'embarqué et désormais dans les serveurs cloud, permet d'atteindre un équilibre optimal entre performance et consommation énergétique.

7 Recommandations pour améliorer l'efficience énergétique

Sur la base de nos analyses, nous proposons plusieurs axes d'optimisation pour améliorer l'efficience énergétique du compilateur Deca et des programmes générés.

La première priorité concerne l'optimisation du code généré. Nos tests ont démontré que la réduction des appels de méthode, particulièrement dans les sections critiques du code, permet de diminuer significativement la consommation énergétique. Pour les calculs intensifs, il est recommandé de privilégier les opérations sur des types simples, en remplaçant notamment les calculs en virgule flottante par des opérations sur des entiers lorsque la précision le permet.

La réduction des branchements inutiles s'avère particulièrement efficace, comme l'ont montré nos mesures de taux de branch-misses.

La gestion des ressources matérielles joue également un rôle crucial. Une allocation optimale des registres, couplée à une réduction des opérations de sauvegarde/restauration dans les sections critiques, permet de minimiser les accès mémoire coûteux en énergie. L'utilisation judicieuse de l'option `-r` du compilateur permet d'adapter cette gestion aux besoins spécifiques de chaque programme.

Enfin, nous recommandons la mise en place d'un processus d'évaluation continue des performances énergétiques. Des tests automatisés et des comparaisons régulières avec d'autres architectures, notamment ARM, permettront d'identifier et d'implémenter de nouvelles optimisations au fil du développement du compilateur.

Ces recommandations visent à établir un équilibre optimal entre performance et efficience énergétique, tout en maintenant la flexibilité nécessaire pour s'adapter aux divers besoins des programmes générés.