Analyse de l'impact énergétique

Roboam Guillaume Raballand Cyprien Talbi El Mehdi
Charles-Mennier Matéo Altieri Aubin

22 janvier 2025
Année académique 2024-2025

Contents

1 Introduction

1.1 Contexte du projet et importance de l'efficience énergétique

Dans un contexte où l'impact environnemental du numérique est devenu un enjeu majeur, l'efficience énergétique des logiciels représente un défi crucial pour les développeurs.
Le projet GL s'inscrit dans cette problématique, car ses performances énergétiques ont un double impact : d'une part sur le processus de développement lui-même, et d'autre part sur l'exécution des programmes compilés.

Un compilateur joue un rôle particulièrement important dans l'efficience énergétique des logiciels qu'il produit. Les choix d'optimisation et de génération de code influencent directement la consommation d'énergie des programmes compilés lors de leur exécution. Dans le cas de notre compilateur Deca, les décisions prises lors de la traduction vers le code assembleur ont des répercussions sur le nombre de cycles d'exécution, qui constitue une approximation pertinente de la consommation énergétique sur ce processeur virtuel.

1.2 Objectifs de l'analyse

Le processus de développement d'un compilateur implique de nombreuses phases de tests et de validation qui peuvent elles-mêmes être consommatrices en ressources. L'optimisation de ces processus, sans compromettre la qualité du compilateur, représente un défi supplémentaire dans notre démarche d'efficience énergétique.

Cette analyse vise donc à examiner en détail ces différents aspects, en s'appuyant sur des mesures concrètes de consommation de ressources. Nous étudierons à la fois l'impact des choix de compilation sur l'efficience du code produit et les stratégies mises en œuvre pour optimiser notre processus de développement. Une attention particulière sera également portée à l'extension du compilateur, où la liberté de conception nous permet d'explorer des solutions innovantes pour minimiser l'empreinte énergétique.

2 Méthodologie d'évaluation de la consommation énergétique

2.1 Outils de mesure utilisés (perf stat)

Afin de mesurer la consommation énergétique de notre compilateur, nous avons utilisé la commande `perf stat` au lieu de `/usr/bin/time` afin d'avoir le plus de données possible à analyser.

2.2 Métriques choisies (temps CPU, accès mémoire, cycles IMA)

Nous avons choisi de nous concentrer sur le nombre de cycles essentiellement (donné par `cpu_atom/cycles/` par la commande `perf stat -e cycles`).

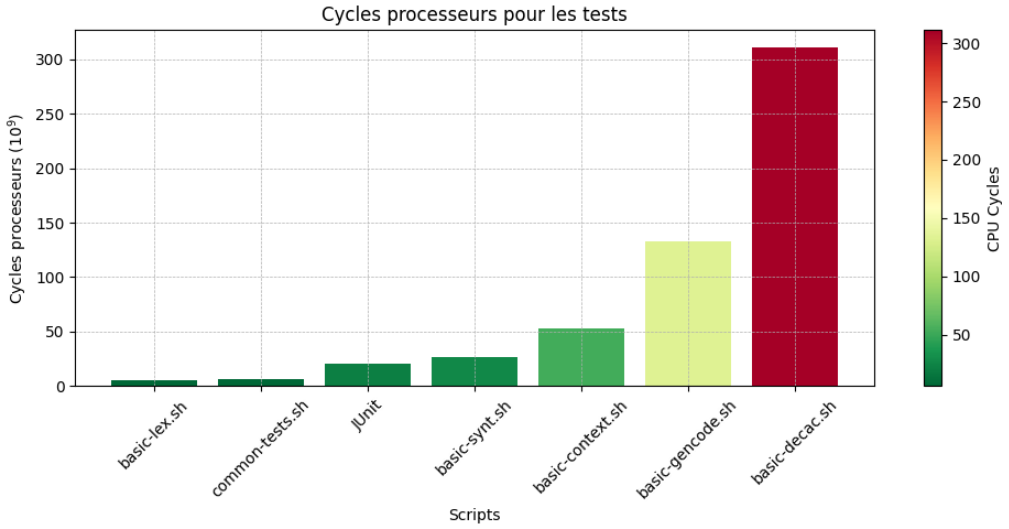
Étant corrélé avec le temps CPU, nous avons voulu éviter de réaliser des tests inutilement pour obtenir des données redondantes.

2.3 Protocole de mesure mis en place

Nous avons mis en place un script Python pour passer les tests et récupérer le nombre de cycles du CPU à chaque fois. Ce dernier est disponible en Annexe, section 7.1.

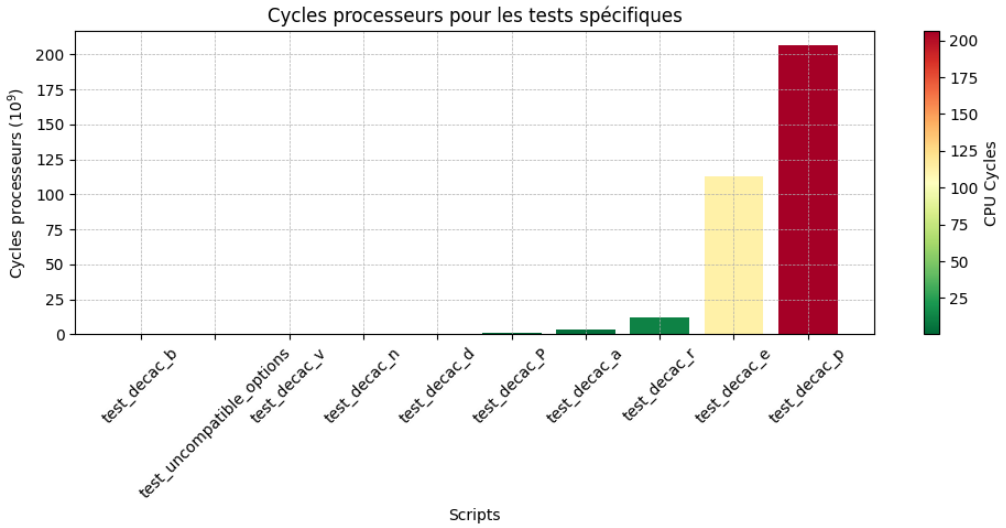
3 Analyse énergétique des tests

Voici un histogramme pour se représenter l'impact de chacun des tests.

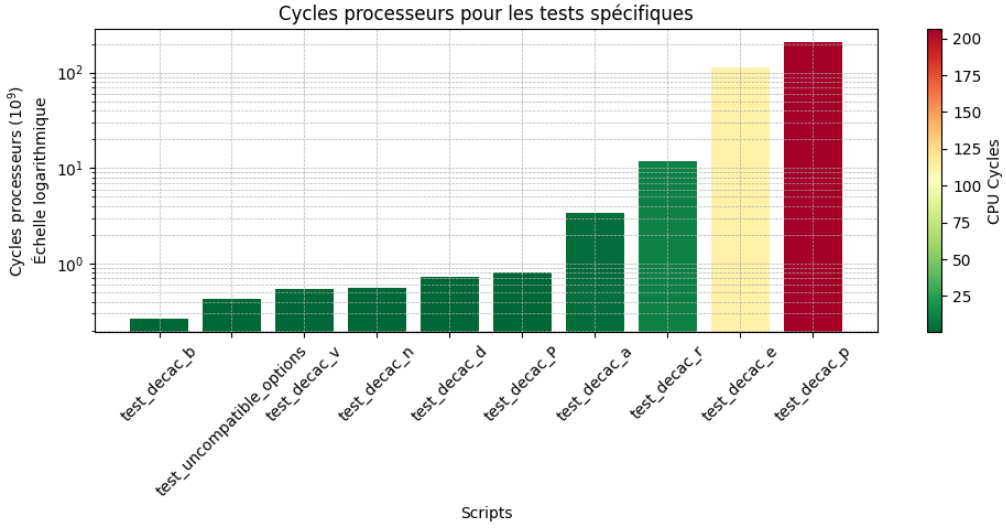


On remarque aisément que le script `basic-decac.sh` est clairement le plus gros consommateur.

Regardons de plus près les cycles :



En échelle logarithmique pour plus de visibilité :



On remarque donc que les 2 tests les plus consommateurs sont les tests de décompilation et ceux pour l'extension Byte.
Ces 2 tests sont particulièrement coûteux car le premier (`decac -p`) écrit beaucoup sur la sortie standard pour vérifier les outputs et permettre un débug

rapide en cas d'échec.
Le deuxième (`decac -e`) doit utiliser la JVM Java pour exécuter le `.class` généré, cela est plus coûteux qu'exécuter IMA.)

4 Optimisations du code généré par le compilateur

Une volonté d'optimisation a été émise dès le début du projet. Les idées que nous avions eues se sont avérées être le contenu entier de l'extension OPTIM et nous avons manqué de temps pour les réaliser.

En rétrospective, nous avons implémenté un compilateur correct mais dont l'optimisation reste à désirer.

5 Optimisation du processus de développement

5.1 Pipeline

Dès le début du projet, nous nous sommes attachés à avoir une pipeline de tests CI/CD automatisés lors d'un push sur une branche.

Pour éviter de faire passer les tests à chaque push, nous avons choisi de lancer les tests lors d'un push uniquement sur notre branche `master` ou `develop`, nos 2 branches principales.

Le runner pour la pipeline était sur un Raspberry Pi 4 Model B, hébergé chez Guillaume. Étant hébergé dans un docker, les ressources étaient donc préservées lorsque nous ne faisions pas de push.

Après le changement de GitHub à GitLab, nous avons essayé de passer la pipeline sur les runners de l'Ensimag, mais sans succès, nous l'avons supprimée car provoquant un échec à chaque fois dû à la mauvaise configuration, ainsi une consommation inutile d'énergie.

6 Impact énergétique de l'extension

6.1 Présentation des choix spécifiques à l'extension

Pour la manipulation et la génération du bytecode, le choix s'est porté sur ASM, une bibliothèque Java réputée pour sa légèreté, sa rapidité et sa puissance dans l'analyse et la transformation du bytecode.

6.2 Analyse comparative des solutions possibles

L'objectif était d'adopter une solution qui permette une manipulation fine du bytecode tout en garantissant une exécution optimisée et un impact énergétique réduit. ASM, par sa conception minimaliste, offre un contrôle direct sur les

instructions du bytecode, évitant ainsi des couches intermédiaires qui pourraient entraîner une surconsommation de ressources.

Plusieurs alternatives à ASM ont été considérées, notamment Byte Buddy et Javassist.

- `Byte Buddy` est connu pour sa facilité d'intégration et son abstraction de haut niveau, ce qui simplifie la génération de code au moment de l'exécution. Cependant, cette abstraction supplémentaire peut entraîner une légère surcharge en termes de performance et d'utilisation des ressources.
- `Javassist`, de son côté, offre une approche orientée objet pour la manipulation du bytecode, ce qui le rend plus simple à utiliser pour certaines applications. Toutefois, sa flexibilité accrue peut engendrer une exécution plus lourde par rapport à ASM.

6.3 Justification des décisions prises

ASM se distingue de ces alternatives par sa faible empreinte mémoire, sa rapidité d'exécution, et sa capacité à offrir un contrôle granulaire, ce qui en fait une solution idéale pour des environnements où l'efficacité énergétique est critique. Le choix d'ASM a été justifié par plusieurs critères clés :

- **Efficacité énergétique** : Grâce à son approche minimaliste et son exécution directe sans couches intermédiaires, ASM permet de réduire la consommation d'énergie en minimisant les cycles CPU nécessaires à la transformation du bytecode.
- **Performance optimisée** : ASM offre des fonctionnalités avancées tout en étant l'une des bibliothèques les plus rapides pour la manipulation du bytecode, ce qui garantit des performances optimales.
- **Adaptabilité** : La flexibilité et la compatibilité d'ASM avec divers environnements permettent de l'utiliser efficacement pour une large gamme d'applications nécessitant une gestion fine du bytecode.

Bien que des comparaisons directes de l'efficacité énergétique entre ASM et d'autres bibliothèques soient limitées, l'accent mis par ASM sur la performance suggère qu'elle est bien adaptée aux applications où l'efficacité est primordiale. En effet, ASM est utilisé dans plusieurs projets notamment ByteBuddy qu'on avait mentionné comme librairie alternative.

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form. ASM provides some common bytecode transformations and analysis algorithms from which custom complex transformations and code analysis tools can be built. ASM offers similar functionality as other Java bytecode frameworks, but is focused on performance. Because it was designed and implemented to be as small and as fast as possible, it is well suited for use in dynamic systems (but can of course be used in a static way too, e.g. in compilers).

ASM is used in many projects, including:

- the **OpenJDK**, to generate the lambda call sites, and also in the Nashorn compiler,
- the **Groovy** compiler and the **Kotlin** compiler,
- **Cobertura** and **Jacoco**, to instrument classes in order to measure code coverage,
- **Byte Buddy**, to dynamically generate classes, itself used in other projects such as **Mockito** (to generate mock classes),
- **Gradle**, to generate some classes at runtime.

7 Annexes

7.1 Programme de calcul de cycles

```python
import os
import subprocess
import re
import matplotlib.pyplot as plt


def parse_result(output: str) -> None | int:
    regex_cmd = r"(\s+(\d+\s+)+)cpu_atom/cycles/"
    match = re.search(regex_cmd, output)
    if match:
        cpu_cycles = match.group(1)
        cpu_cycles = int(cpu_cycles.replace("\u202f", "")
                                   .strip()
                                   )
        return cpu_cycles
    else:
        return None


def get_cycles(file_path: str,
               executable_path: None | str = None
               ) -> int:
    try:
        if executable_path:
            result = subprocess.run(
                ["perf", "stat", "-e", "cycles",
                executable_path, file_path],
                capture_output=True,
                text=True,
            )
        else:
            result = subprocess.run(
                ["perf", "stat", "-e", "cycles", file_path],
                capture_output=True,
                text=True,
            )
    except subprocess.CalledProcessError as e:
        print(f"Failed to execute {file_path}: {e}")
        exit(1)
    return parse_result(result.stderr)


def execute_files_recursively(folder_path: str,
                              executable_path: str
                              ) -> None:
```

```
    result_list = []
    for root, dirs, files in os.walk(folder_path):
        for file in files:
            file_path = os.path.join(root, file)
            res = get_cycles(file_path=file_path,
                             executable_path=executable_path
                                 )
            result_list.append(res)


def get_general_tests_results() -> None:
    res_list = []
    scripts = [
        "basic-lex.sh",
        "basic-synt.sh",
        "basic-context.sh",
        "basic-gencode.sh",
        "basic-decac.sh",
        "common-tests.sh",
    ]
    for script in scripts:
        print("Executing script: ", script)
        filepath = "src/test/script/" + script
        res = get_cycles(file_path=filepath)
        res_list.append({script: res})
    # res_list = [
    #     {"basic-lex.sh": 5846973010},
    #     {"common-tests.sh": 6722139564},
    #     {"JUnit": 20450162247},  # Fait manuellement
    #     {"basic-synt.sh": 26910272860},
    #     {"basic-context.sh": 52994838570},
    #     {"basic-gencode.sh": 133342103418},
    #     {"basic-decac.sh": 311182213553},
    # ]

    scripts = [list(d.keys())[0] for d in res_list]
    cycles = [list(d.values())[0] / 10**9 for d in res_list]

    norm = plt.Normalize(min(cycles), max(cycles))
    colors = plt.cm.RdYlGn_r(norm(cycles))

    fig, ax = plt.subplots(figsize=(10, 5))
    ax.bar(scripts, cycles, color=colors)
    ax.set_xlabel("Scripts")
    ax.set_ylabel("Cycles processeurs ($10^9$)")
    ax.set_title("Cycles processeurs pour les tests")
    ax.set_xticks(range(len(scripts)))
    ax.set_xticklabels(scripts, rotation=45)
    ax.grid(True, which="both", linestyle="--", linewidth
        =0.5)
```

```
    plt.tight_layout()
    plt.colorbar(
        plt.cm.ScalarMappable(
                        norm=norm,
                        cmap="RdYlGn_r"),
        label="CPU Cycles",
        ax=ax
    )
    plt.show()


def get_specific_tests_results() -> None:
    # Do manually for each script
    # modyfing the basic-decac.sh script each time
    res_list = [
        {"test_uncompatible_options": 431817939},
        {"test_decac_b": 265736457},
        {"test_decac_p": 206584713700},
        {"test_decac_v": 537351265},
        {"test_decac_n": 556152940},
        {"test_decac_r": 11807256673},
        {"test_decac_d": 721265054},
        {"test_decac_P": 812773729},
        {"test_decac_a": 3372825496},
        {"test_decac_e": 112588788429},
    ]
    res_list.sort(key=lambda x: list(x.values())[0])
    # function = "test_decac_e"
    # cycles_list = []
    # for _ in range(5):
    #     res = get_cycles(file_path="src/test/script/basic-decac.sh")
    #     cycles_list.append(res)
    # mean_cycles = sum(cycles_list) / len(cycles_list)
    # print({function: mean_cycles})
    # exit()

    scripts = [list(d.keys())[0] for d in res_list]
    cycles = [list(d.values())[0] / 10**9 for d in res_list]

    norm = plt.Normalize(min(cycles), max(cycles))
    colors = plt.cm.RdYlGn_r(norm(cycles))

    fig, ax = plt.subplots(figsize=(10, 5))
    ax.bar(scripts, cycles, color=colors)
    ax.set_xlabel("Scripts")
    ax.set_ylabel("Cycles processeurs ($10^9$)")
    ax.set_title("Cycles processeurs pour les tests specifiques")
    ax.set_xticks(range(len(scripts)))
```

```
    ax.set_xticklabels(scripts, rotation=45)
    ax.grid(True, which="both", linestyle="--", linewidth
        =0.5)
    plt.tight_layout()
    plt.colorbar(
        plt.cm.ScalarMappable(
                    norm=norm,
                    cmap="RdYlGn_r"),
        label="CPU Cycles",
        ax=ax
    )
    plt.show()


def main() -> None:
    get_general_tests_results()
    get_specific_tests_results()


if __name__ == "__main__":
    main()
```