Analyse énergétique

Afin de limiter l'impact énergétique de notre projet, nous avons mis en œuvre des stratégies pouvant être regroupées en deux grandes catégories : la limitation de notre utilisation d'énergie pendant le développement du compilateur, et l'optimisation des programmes assembleurs générés par le compilateur afin qu'ils soient plus efficaces qu'une approche purement naïve.

Nous étudions ces deux aspects successivement dans ce document.

Optimisation du processus de développement

Lors du développement de notre compilateur, nous avons naturellement dû le tester de la façon la plus complète possible, ce qui a nécessairement impliqué un coût énergétique. Cependant, des mesures ont été prises pour raisonnablement limiter ce coût lorsque cela était possible.

Tout d'abord, bien que nous disposions de la possibilité de lancer d'un coup l'ensemble de nos tests (les compilant et les exécutant, en assembleur ima comme en bytecode Java), nous l'avons utilisée de façon parcimonieuse, lui préférant, à chaque fois que cela était suffisant, la compilation du seul test que nous tentions, à ce moment-là, de faire fonctionner correctement.

Par ailleurs, lorsqu'il s'agissait de corriger des problèmes à l'exécution du programme (c'est-à-dire des problèmes de génération de l'assembleur), nous avons également eu une approche raisonnable consistant à modifier directement le code en assembleur pour tester des corrections, avant de les retranscrire dans notre compilateur directement (pour éviter d'innombrables compilations de notre compilateur, suivies de compilations et d'exécutions incessantes d'un test).

Par ces points de raisonnable responsabilité, nous avons tenté de limiter modestement la consommation énergétique de notre processus de développement. Mais notre approche vis-à-vis de l'analyse énergétique ne s'est naturellement pas arrêtée là.

Optimisation des programmes en assembleur générés par le compilateur

Bien que nous n'ayons pas choisi l'extension d'optimisation, nous avons cherché à réduire les opérations inutiles effectuées dans les programmes en assembleur générés par notre compilateur.

Notre approche s'est concentrée sur les opérations facilement éliminables à la compilation, lorsque ni variable, ni méthode n'entre en compte. Cette stratégie nécessite une évaluation de certaines expressions à la compilation, mais garantit que ces mêmes expressions n'auront plus jamais besoin d'être évaluées à l'exécution (alors qu'elles auraient dû l'être au moins une fois par exécution, voire bien davantage dans une instruction contenue dans une boucle while ou une méthode).

Premièrement, notre compilateur simplifie les branchements conditionnels lorsque la condition ne contient ni appel de méthode, ni utilisation de variable.
Par exemple, ce code :

```
if (0 == 0) {
        //instruction 1
}
else {
        //instruction 2
}
```

sera compilé comme s'il s'agissait réellement du code :

```
//instruction 1
```

En termes d'assembleur, nous simplifions donc l'expression compliquée suivante :

```
LOAD #0, R1
```

CMP #0, R1
BNE else
if:
//instruction 1
BRA endif
else:
//instruction 2
endif:

En l'expression simple :

//instruction 1

Soit un passage, dans le pire des cas, de 19 à 0 cycles nécessaires à l'exécution pour la structure conditionnelle.

L'autre point d'optimisation concerne les opérations (de toute nature) ne comprenant ni variable, ni méthode, qui sont effectuées directement à la compilation (plutôt qu'au moins une fois par exécution, voire bien davantage).

Par exemple, l'instruction :

*(52 * 10 / 13) % 3*

(par exemple dans *print((52 * 10 / 13) % 3)*)

sera compilée simplement comme :

1

Autrement dit, en assembleur, on passe de :

LOAD #52, R1
MUL #10, R1
QUO #13, R1
REM #3, R1

à :

(éventuellement LOAD #1, R1 si le résultat de l'opération est utilisé par la suite, par exemple dans un print ou dans une déclaration de variable)

Ce qui revient à un passage de 126 cycles à seulement 4 (voire 0), une amélioration conséquente.

Il est à noter que d'autres expressions sont également partiellement simplifiées, même si elles contiennent des appels de méthode ou des variables, comme :

```
x + 30 / 5 - 2
```

qui deviendra :

```
x + 4
```

Soit un passage de 60 cycles à seulement 6 cycles.

Enfin, il est bien entendu possible de combiner ces deux optimisations, en passant de :

```
if ((52 * 10 / 13) % 3 == 1) {
    //instruction 1
}
else {
    //instruction 2
}
```

à :

```
//instruction 1
```

Ceci simplifiant :

```
LOAD #52, R1
MUL #10, R1
QUO #13, R1
REM #3, R1
CMP #1, R1
BNE else
if:
//instruction 1
BRA endif
else:
//instruction 2
endif:
```

En :

```
//instruction 1
```

Ce qui revient à une diminution de temps d'exécution de la structure conditionnelle de 141 cycles à 0.

À toutes ces diminutions de nombre de cycles nécessaires correspond naturellement une diminution du coût énergétique des programmes à l’exécution.