Documentation sur le bilan énergétique du projet

GL37
Thomas Avare, Quentin Candaele, Helena Cazals, Germain Vu, Andry Rakotonjatovo

22 janvier 2022

"Ce qui caractérise notre époque, c'est la perfection des moyens et la confusion des fins."
- Albert Einstein

Table des matières

L'informatique est de plus en plus présent dans notre société, les nouvelles technologies et les objets IOT (Internet Of Things) sont omniprésents dans nos vies. Même si beaucoup de gens n'ont pas les connaissances techniques nécessaires pour se rendre compte de l'impact énergétique, que cela soit lors de l'utilisation d'un produit informatiques ou encore lors de sa conception, celui-ci a une part essentielle dans toutes ces phases. Mais en tant que futurs ingénieurs en informatique, il nous est important d'apprendre et de développer notre conscience écologique. Les générations contemporaines d'informaticiens développant sur des machines très peu limités techniquement. La manière de développer est très différentes de celles de nos aînés, qui devaient se soucier des limitations techniques, telles que le nombres de registres, la mémoire vive, les limitations des langages, etc... Le projet GL est donc un bon moyen de se rendre compte de ce que l'on peut faire peu pour faire beaucoup avec des limitations, dans notre cas via la machine virtuelle cible IMA fourni au préalable.

1 EFFICIENCE DU PROCÉDÉ DE FABRICATION

Du au manque de temps, nous n'avons pas pu optimiser au maximum notre compilateur et autant que voulu dans le temps. Nous avons privilégié le fait que notre projet soit le plus fonctionnel possible et ce jusqu'à la soutenance. Globalement, nous avons fait attention à essayer d'utiliser le moins de registre possible sur toutes les opérations et aussi à libérer les registres qui ne vont plus être utilisés lors des cycles suivants.

D'autre part, nous nous sommes débarrasser au maximum des parties de codes dîtes mortes, sur la fin du projet, la couverture du code effectuée à l'aide de l'outil Jacoco nous a permis d'évaluer que nous parcourions 76% de notre code à la suite des phases et certaines parties du code n'étant jamais exploré malgré la création de nouveaux tests étant censé rentrer dans ces parties de codes, celle-ci n'était toujours pas exploré et ont été jugée inutile (exemple figure 1).

```
221.    public void verifySignature(DecacCompiler compiler, EnvironmentExp localenv, ClassDefinition currentclass,
222.                                Signature signature) throws ContextualError {
223.        LOG.debug("verify Signature: start");
224.        Signature realsignature = compiler.getDefinition(getName()).asMethodDefinition("Pas une méthode",
225.                this.getLocation()).getSignature();
226.        if (realsignature.size() != signature.size()) {
227.            throw new ContextualError("signature non conforme, pas le bon nombre de paramètres", this.getLocation());
228.        }
229.        if (realsignature.size() == 0 && signature.size() == 0) {
230.            LOG.debug("verify Signature: end");
231.            return;
232.        }
233.        int size = realsignature.size();
234.        for (int cpt = 0; cpt < size; cpt++) {
235.            if (!signature.paramNumber(cpt).sameType(realsignature.paramNumber(cpt))) {
236.                throw new ContextualError("Le param " + cpt + " est de type " + signature.paramNumber(cpt).getName().getName()
237.                        + " alors qu'il devrait être de type " + realsignature.paramNumber(cpt).getName().getName(),
238.                        this.getLocation());
239.            }
240.        }
241.        LOG.debug("verify Signature: end");
242.    }
```

FIGURE 1 – exemple de code mort

Le code mort peut sembler être un détail mais à la compilation c'est une partie de code

en plus en plus à générer et sur des applications à grande échelle, cela peut devenir une différence certe petite mais non négligeable.

Pour l'extension, afin de faire des économies énergétiques, nous avons fait comme précédemment pour le tester. uniquement des tests utiles mais nous également en grande partie uniquement du code utile et factorisé au maximum. D'autre part, nous avons passé beaucoup de temps à étudier les différents algorithmes de manière la plus théorique afin de ne pas tester inutilement.

Les algorithmes ont aussi été optimisés au maximum afin de faire converger le plus rapidement nos différentes fonctions pour éviter de repasser inutilement dans des boucles afin de limiter au maximum le nombre de cycles.

Certaines options peuvent faire également économiser de l'énergie par une diminution du nombre de cycle au sein de la machine cible `IMA`. Par exemple l'option no-check qui va enlever les tests de division par 0, les débordement arithmétiques sur les flottants, l'absence de return lors de l'exécution d'une méthode, les conversions de types impossibles et les déréférencement de null pour les programmes incorrects et les débordement de mémoire (pile ou tas) pour les pour les programmes corrects.

Ainsi on peut grandement réduire le nombre de cycle nécessaire et obtenir un résultat correct sur de nombreux programmes si l'on sait d'avance qu'ils sont corrects et réduire le nombre de cycle nécessaire pour sortir des erreurs sur des programmes incorrects.

Une autre option permettant de réduire la consommation énergétique et l'utilisation de `Thread` permettant d'exploiter au maximum les propriétés de compilation java et exploiter au maximum les différents CPU et les temps d'exécution au sein de ces CPU.

2 Efficience du produit fini

Le Produit fini fonctionne (actuellement à posteriori du rendu) et n'est malheureusement pas forcément efficace mais il est en grande partie fonctionnel. Notre plus grand regret a été, comme dit au préalable le manque de temps qui ne nous a pas forcément permis d'optimiser au maximum notre compilateur et donc est assez énergivore lorsque l'on regarde comparé à d'autres équipes dans les classement du palmarès.

Voici Alors quelques résultats fournis par GNU-time lors de l'exécution de différents programmes deca fournis.

Pour plus d'information à propos de GNU-time : `https://www.gnu.org/software/time/`

```
Command being timed: "decac src/test/deca
/codegen/perf/provided/syracuse42.deca"

User time (seconds): 0.58
System time (seconds): 0.21
Percent of CPU this job got: 110%
Elapsed (wall clock) time (h:mm:ss or m:ss):
0:00.71
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 49748
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 5346
Minor (reclaiming a frame) page faults: 10870
Voluntary context switches: 555
Involuntary context switches: 2131
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 1
Page size (bytes): 4096
Exit status: 0
```

```
Command being timed: "decac -n src/test/deca
/codegen/perf/provided/syracuse42.deca"

User time (seconds): 0.56
System time (seconds): 0.23
Percent of CPU this job got: 99%
Elapsed (wall clock) time (h:mm:ss or m:ss):
0:00.81
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 49236
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 5331
Minor (reclaiming a frame) page faults: 10714
Voluntary context switches: 652
Involuntary context switches: 2602
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 1
Page size (bytes): 4096
Exit status: 0
```

Compilation de `syracuse42.deca` avec et sans l'option no-check.

```
Command being timed: "ima src/test/deca/
codegen/perf/provided/syracuse42.ass"

User time (seconds): 0.01
System time (seconds): 0.01
Percent of CPU this job got: 71%
Elapsed (wall clock) time (h:mm:ss or m:ss):
0:00.03
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 1664
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 124
Minor (reclaiming a frame) page faults: 2398
Voluntary context switches: 18
Involuntary context switches: 77
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 1
Page size (bytes): 4096
Exit status: 0
```

```
Command being timed: "ima src/test/deca/
codegen/perf/provided/syracuse42.ass"

User time (seconds): 0.01
System time (seconds): 0.01
Percent of CPU this job got: 78%
Elapsed (wall clock) time (h:mm:ss or m:ss):
0:00.03
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 1636
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 124
Minor (reclaiming a frame) page faults: 2396
Voluntary context switches: 23
Involuntary context switches: 78
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

On a pris différentes mesures. Deux de compilation dont une avec l'option no-check d'activé et respectivement la compilation des deux programmes.

à la compilation, il y a quelques différences notables intéressantes, notamment le temps de compilation (user time) qui est très légèrement différent et inférieur lorsque no-check est activé. Le système time quant à lui est plus long lorsque l'option no-check est activé. Une autre partie qui nous a semblé cohérente de surligner est la partie major "Major page faults" qui mesure le nombre d'entrée et sortie (I/O). On voit qu'il est inférieur pour la compilation sous non-check.

Bien évidemment on remarque le pourcentage de CPU utilisé qui passe 110% à 99% lors

de l'activation du no-check.

Quant à la partie interprétation par la machine ima, Il n'y a pas vraiment de différence, à part la part de CPU utilisée et les "minor page faults", correspondant à une exception ou erreur dans le plan d'adressage mineur et donc pas très grave. Pour la compilation, on se référera à l'interprétation avec l'option -s de la machine IMA.

On a les résultats suivants :

— sans no-check : `Nombre d'instructions : 73 Temps d'exécution : 2313`

— avec no-check : `Nombre d'instructions : 69 Temps d'exécution : 2271`

On voit que le nombre d'instruction et le temps d'exécution est inférieur pour la compilation avec no-check.

3 Conclusion

notre compilateur n'est sûrement pas le plus énergivore, mais ce n'est pas non plus le moins énergivore également. De nombreux points peuvent-être amélioré, d'un point de vue du code mais aussi énérgétique.