PROJET GL

ANALYSE DES IMPACTS ÉNERGÉTIQUES

Grenoble INP-Ensimag

Amine ANGAR
Dy EL ALEM
Mouad TARBOUI
Oussama KADDAMI
Youness LEHDILI

INTRODUCTION:

L'impact énergétique était inscrit au cœur de notre démarche de conception et de tests depuis le début du projet. Ainsi nous avons veillé à concevoir les structures les plus optimales en terme de consommation de l'énergie et ainsi agir en tant que professionnels responsables. Étant donné que les deux premières parties étaient plus contraintes que la partie C et encore l'extension par le cahier de charge, nos optimisations concernaient plutôt ces parties-là.

Optimisation des algorithmes et structures de données durant la partie C:

Tout au long du projet, notre conception des algorithmes et codes prenait en considération l'impact énergétique.

Fonctions de Génération du code:

Pendant notre conception de la partie sans objet, notre conception nous obligeait de définir une fonction codeExp prenant en paramètre le compiler et un registre, en plus de la fonction codeGenInst qui était elle aussi nécessaire puisque la fonction codeGenInst appelait pour chaque instruction cette dernière fonction. Pendant notre phase de conception durant l'incrément "Langage essentiel", nous avons décidé d'optimiser le code et d'éliminer cette duplication. Nous avons donc à l'issue de cette décision passer un temps significatif à restructurer tout notre code et refaire les tests pour s'assurer du bon fonctionnement de la version mise à jour.

Instruction assembleur:

Tout au long du projet, nous avons essayé de repérer et d'optimiser les instructions assembleurs. Nous avons donc décidé d'éliminer les instructions CMP après une instruction qui positionnent les bits de comparaison comme STORE et LOAD.

Évaluation des conditions:

Les expressions qui produisent des résultats booléens ont un comportement différent des autres. Pour ces dernières, nous définissons en plus de la fonction codeGenInst nécessaire ,par exemple dans le cas d'une affectation du résultat d'une expression booléenne à une

variable du même type, nous avons défini une fonction codeCond qui permet d'évaluer directement l'expression booléenne et instaurer un branchement conditionnel à l'étiquette associée à la fonction selon la valeur booléenne de l'expression à évaluer. Cela nous fait une économie donc des instructions associées à stocker le résultat de l'évaluation dans un registre et le charger ensuite pour l'évaluer.

D'autre part, les évaluations des expressions booléennes reviennent à un simple branchement inconditionnel à l'étiquette donc selon la valeur booléenne de l'expression. Cela nous a évité donc les instructions assembleur d'évaluation de l'expression.

```
@Override
public
void codeGenInst(DecacCompiler compiler) {
    if (value) {
        compiler.addInstruction(new LOAD(new ImmediateInteger(1),
                Register.getR(getRP(compiler))));

    }
    else {
        compiler.addInstruction(new LOAD(new ImmediateInteger(0),
                Register.getR(getRP(compiler))));

    }
}
```

Utilisation de fonctions:

Pendant notre passage de l'incrément "Sans Objet" à l'incrément "Langage essentiel", nous avons essayé d'adapter les fonctions utilisées pour la génération du code de la fonction main, pour une utilisation globale. Nous avons donc fait des adaptation pour que la génération du code des méthodes des classes et des fonctions d'initialisation se fait avec les mêmes fonctions qui évaluent les instructions et la déclaration des variables de la fonction principale.

D'autre part, nous avons essayé de factoriser le code assembleur que génère notre compilateur pour certaines instructions. Par exemple, nous avons défini une fonction instanceOf qui permet de gérer tous les appels à instanceOf et ainsi le cast aussi sans avoir besoin de dupliquer à chaque fois le code assembleur associé.

Un peu d'algorithmique:

Nous avons veillé à la bonne application des principes d'algorithmiques étudiés pendant nos deux années à l'ensimag afin d'optimiser le code produit et ainsi la consommation d'énergie intrinsèque. Cela commence, par exemple, tout simplement par une évaluation des vecteurs dans l'ordre canonique. Dans notre cas, nous privilégions à coup sûr les itérations de la sorte:

```
for (AbstractDeclVar var : getList()) {
    //codegenVar
    var.codeGenAndLinkDeclVariableMain(compiler);
}
```

aux itérations par indices non optimales.

Nous avons aussi essayé d'utiliser les ressources de la machine de façon optimale à savoir l'exploitation de la mémoire cache, ainsi nous avons essayé d'utiliser au mieux la localité spatiale et temporelle, ainsi que les principes de memoïsation.

Structures de données utilisées :

Nous avons essayé de choisir les structures de données les plus compatibles avec notre cas d'utilisation. Ainsi, nous avons décidé d'utiliser par exemple, une structure FIFO(linkedList), pour la sauvegarde et la restitutions de certaines valeurs lors des appels aux fonctions. Ceci permet donc une insertion et une suppression $0(1)$. Après la bonne compréhension, des étapes nécessaires pour l'exécution de l'instanceof et du cast qui utilise cette dernière, nous avons décidé d'éviter l'utilisation du downCast et d'essayer de faire des redéfinitions des fonctions concernés ce qui est résolu facilement lors de la phase de la génération de la table de méthodes associée à la classe.

Optimisation des algorithmes et structures de données durant Extension

Les factorielles et les puissance sont particulièrement lourdes en multiplications, Nous avons cherché à éliminé leur calcul, on remarque dans la série entière du cosinus que la factoriel dans le dénominateur augmente de 2 en 2,on peut donc utiliser la technique de mémorisation afin de conserver la valeur du dénominateur et la multiplier a chaque itération p par (2p)(2p+1), pour le numérateur , la puissance augmente de 2 en 2, et le signe change à chaque itération , il suffît de conserver le numérateur et le multiplier par x^2

	intervalle	nombre d'erreur	erreur moyenne en ulp	pas	nombres de tests	temps de calcul
cos	[0;0.785]	7820	2 ulp	2^{-22}	3.294.199	0m4,062s
cos	[0.785;1.047]	8730	2 ulp	2^{-22}	1.098.066	0m1,572s
cos	[1.047;2.3561]	210135	15 ulp	2^{-22}	5.490.331	0m8,434s
cos	[2.3561;3.141]	210135	2 ulp	2^{-22}	4.290.191	0m5,870s
cos	[3.141;3.926]	1031825	2.31 ulp	2^{-22}	6.588.398	0m6,099s
cos	[3.926;4.5]	457529	5.35 ulp	2^{-18}	600844	0m0,441s
cos	[4.5;6.28]	276015	28.43 ulp	2^{-18}	466617	0m0,762s
ulp	[0;1]	0	0 ulp	2^{-15}	32.768.000	0m0,173s
ulp	[0;1.000]	0	0 ulp	2^{-18}	128000	0m0,196s
ulp	[1.000;1.000.000]	0	0 ulp	2^{-15}	7992000	0m1,294s
atan	[0;1.000]	0	2 ulp	2^{-15}	1.024.000	0m2,365s
atan	[1.000;100.00]	0	1 ulp	2^{-18}	3.168.000	0m0,974s
asin	[−1;1]	8042	2 ulp	2^{-22}	8.388.608	0m22,782s
asin	[0.72;1]	4939	2 ulp	2^{-22}	73401	0m0,283s
asin	[−1;−0.72]	4944	2 ulp	2^{-22}	73401	0m0,344s
asin	[−0.72;0.72]	996	2 ulp	2^{-22}	377488	0m0,163s
sin	[0;0.785]	424	2 ulp	2^{-18}	205.888	0m0,345s
sin	[0.785;1.57]	2904	2 ulp	2^{-18}	205.888	0m0,314s
sin	[1.57;2.35]	34729	2.16 ulp	2^{-18}	205.888	0m0,374s
sin	[2.35;3.141]	154329	26 ulp	2^{-22}	205.888	0m0,343s
sin	[3.141;3.926]	115609	46.8 ulp	2^{-22}	204.055	0m0,353s
sin	[3.926;5.49]	70438	2.17 ulp	2^{-18}	409994	0m0,543s
ssin	[5.49;6.28]	186321	40.4 ulp	2^{-18}	207094	0m0,349s