GRENOBLE INP UGA Ensimag

Extension Optim

Code faster, not harder.

GL22

January 25, 2024

Contents

List of Code Listings

1 Introduction

La plupart des compilateurs existants, tels que GCC[1] , ne se contentent pas de convertir un code en entrée en langage machine, mais ils s'efforcent également d'optimiser le code afin de réduire sa consommation d'énergie et d'améliorer son temps d'exécution. Au fil des années, cette fonctionnalité s'est révélée être un atout considérable. L'optimisation de code est un outil puissant, bien que peu de gens en aient pleinement conscience ou même en connaissent l'existence. Sa puissance est telle qu'elle peut susciter la fierté de la rapidité d'exécution, même pour un code initialement mal conçu. Il est

vrai que l'optimisation de code nécessite des traitements supplémentaires qui peuvent entraîner une consommation d'énergie ou un temps d'exécution plus élevé. Cependant, les avantages en termes de vitesse d'exécution du code final sont considérables. Dans le cadre du projet, le choix de l'extension

"otim" revêt une importance particulière. Cette extension, au-delà d'être un simple outil, est une source de motivation continue. Elle offre la possibilité de ressentir une satisfaction personnelle face à la rapidité accrue du code, même dans des situations où la qualité initiale du code pourrait être perfectible. En

fin de compte, l'extension "otim" n'est pas seulement un moyen d'optimiser le code, mais une passerelle vers l'amélioration constante et l'accomplissement personnel à travers la programmation.

2 Algorithmes

Nous avons introduits plusieurs algorithmes pour optimiser le code.

2.1 Strength Reduction

La "Strength Reduction"[2] vise à remplacer des opérations coûteuses par des opérations moins coûteuses tout en préservant la sémantique du programme. Dans le cas de la multiplication ou de la division par des puissances de 2, cela peut être réalisé par des opérations de décalage (shift) plutôt que des

[1] https://fr.wikipedia.org/wiki/GNU_Compiler_Collection
[2] https://en.wikipedia.org/wiki/Strength_reduction

multiplications ou divisions, ce qui est généralement plus efficace en termes de cycles CPU.

2.1.1 Principe

```
{
    int x = 1;
    x = x * 64;
    x = x / 2;
}
```

Avant le Stength reduction

```
{
    int x = 1;
    // Mutiplication deviens des shift à gauche
    x = x << 6;
    // Division deviens des shift à droite
    x = x >> 1;
}
```

Après le Stength reduction

2.1.2 Limitation

Maintenant, explorons les limitations de l'algorithme de réduction de puissance. Avant d'aborder ces points, soulignons tout d'abord un de ses atouts:

Son intégration harmonieuse avec d'autres algorithmes, en particulier la propagation et le pliage de constantes. Après l'exécution de la propagation et du pliage de constantes, il devient aisé de détecter les puissances de 2 pour effectuer des opérations de décalage (*shift*).

Cependant, il est essentiel de noter que la machine virtuelle IMA impose des restrictions. Les opérations de décalage d'un bit à la fois, bien que bénéfiques, ont un coût cyclique de 2 cycles. En revanche, l'opération de multiplication (*MULT*) a un coût bien plus élevé, soit 20 cycles. Bien que cela puisse sembler excessif, les avantages de l'algorithme de réduction de puissance s'atténuent lorsque l'on dépasse les puissances de 10. Dans de

tels cas, l'algorithme perd de son intérêt et alourdit le programme en termes de temps d'exécution et d'utilisation de la mémoire. Dans le cadre de notre extension, nous avons pris la décision de ne pas dépasser les puissances de 10 pour l'application de l'algorithme. Cette restriction permet de maintenir un équilibre entre l'efficacité de l'algorithme et les performances globales du programme.

2.2 Propagation & Constant Folding

Constant Folding[3] est une technique d'optimisation qui consiste à évaluer des expressions contenant des constantes au moment de la compilation plutôt qu'à l'exécution. L'idée est de calculer le résultat des opérations lorsque les valeurs des opérandes sont connues à l'avance, et de remplacer l'expression originale par sa valeur constante.

Par exemple, considérons l'expression arithmétique suivante :

```
int x = 42 * 2;
```

Avec Constant Folding, le compilateur évaluera cette expression pendant la compilation.

La Propagation[4] est une autre technique d'optimisation qui vise à substituer les valeurs constantes à travers le code source. L'idée est de propager la valeur d'une variable constante dès que son affectation est connue.

Considérons l'exemple suivant en Deca :

```
int x = 777;
int y = x * 64;
```

Avec la "Constant Propagation", le compilateur propagera la valeur constante de x à la variable y lors de la compilation.

[3] https://en.wikipedia.org/wiki/Constant_folding

[4] https://cran.r-project.org/web/packages/rco/vignettes/opt-constant-propagation.html

2.2.1 Integration au code

Nous avons introduit deux étapes d'optimisation supplémentaires, B'1 et B'2, dans notre processus de compilation, les positionnant juste après l'étape B du programme. Ces étapes, spécifiques à l'optimisation, ont été intégrées de manière ingénieuse pour améliorer les performances du code généré.

La première passe, B'1, a été conçue de manière à être exécutée avant l'élimination du code mort. Cette approche permet une meilleure détection des parties du code qui ne peuvent pas être atteintes. Après l'élimination du code mort, il devient possible d'obtenir des informations plus précises sur les valeurs des variables, ce qui facilite la mise en œuvre de la propagation de constantes dans la deuxième passe, B'2.

L'élimination du code mort, effectuée après B'1, contribue à simplifier le code et à rendre plus clair le flux d'exécution du programme. Cette clarification du code facilite la détection des valeurs constantes pour la propagation dans la passe suivante.

La deuxième passe, B'2, est dédiée à la propagation de constantes. Cette optimisation consiste à substituer des valeurs constantes partout où cela est possible, améliorant ainsi l'efficacité du code généré. La détection des valeurs constantes après l'élimination du code mort permet une propagation plus précise et plus étendue.

```
if (compilerOptions.getOPTIM()) {
    // pass 1 of OPTIM
    prog.ConstantFoldingAndPropagation(this);

    // pass 2 of OPTIM
    prog.DeadCodeElimination();

    // pass 3 of OPTIM
    isPass3 = true;
    prog.ConstantFoldingAndPropagation(this);
}
```

Listing 1: Program Deca après Execution du Strength Reduction

2.3 Liveness Analysis

Dans le cadre de notre projet, nous avons mis en œuvre une analyse de "Live Variables" avec une portée spécifique, se limitant aux zones critiques du programme, en particulier aux boucles "while". L'objectif principal de cette approche est d'optimiser l'utilisation des registres en minimisant les accès à la pile, une opération coûteuse en termes de cycles CPU.

L'analyse de "Live Variables" est une technique courante en compilation visant à déterminer les points du programme où les variables sont utilisées ou "vivantes". Dans notre cas, nous avons adapté cette analyse pour se concentrer exclusivement sur les zones de code sensibles, les hotspots, identifiés comme étant les boucles "while".

2.3.1 Pourquoi Limiter l'Analyse aux Boucles While ?

Les boucles "while" sont souvent des points critiques dans un programme, contribuant significativement au temps d'exécution. Ces boucles peuvent contenir des opérations répétitives, et optimiser leur performance peut avoir un impact global sur le programme. En mettant en œuvre l'analyse de "Live Variables" spécifiquement pour ces boucles, nous cherchons à maximiser l'efficacité des registres, en évitant au maximum les accès à la pile pendant l'exécution des boucles.

2.3.2 Stratégie d'Optimisation : Allocation de Registres

Notre approche consiste à allouer des registres aux variables et aux objets présents dans les boucles "while". En plaçant ces éléments dans des registres, nous réduisons la nécessité d'accéder à la mémoire, améliorant ainsi les performances du programme. Les registres offrent un accès plus rapide aux données que la pile, contribuant ainsi à minimiser les cycles CPU consacrés à ces opérations.

Nous avons utilisé une *HashedSet* afin de stocker les live variables des boucles while:

```
class While {

    private final HashSet<AbstractIdentifier> liveVariables = new HashSet<>();
```

```
}

class AbstractIdentifier{
    @Override
    public int hashCode() {
        return getName().hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof AbstractIdentifier)
            return ((AbstractIdentifier) obj).getName().equals(getName());
        return false;
    }

}
```

Du coup l'idée est de simplement stocker les variables dans des registres et de les remettre dans la piles vers la fin de la boucle:

```
            if (!isInWhile)
                loadLiveVariable(compiler); // <-- Là c'est les load

            // Start + Body
            compiler.addLabel(startOfWile);

            body.codeGenListInst(compiler);

            // while (cond)
            this.condition.codeGenInstOP(compiler);
            compiler.addInstruction(new CMP(
                    0,
                    Register.getR(compiler.getStack().getCurrentRegister() - 1)
            ));
            compiler.getStack().decreaseRegister();


            compiler.addInstruction(new BNE(startOfWile));
            compiler.addLabel(endOfWile);

            //Store variables
            if (!isInWhile)
                storeLiveVariable(compiler); // <-- Là c'est les store
```

2.3.3 Limitations

Dans notre approche d'optimisation, nous avons ciblé la partie la plus consommatrice en termes de ressources du programme, en particulier les boucles "while". Cette stratégie vise à optimiser significativement les performances du code en minimisant les accès à la mémoire, qui peuvent être coûteux en termes de cycles CPU.

L'un des points forts de notre algorithme réside dans sa capacité à cibler spécifiquement les zones critiques du programme, en l'occurrence les boucles "while", et à apporter des améliorations significatives à leur exécution. En concentrant nos efforts sur les parties les plus intensives en ressources, nous avons pu obtenir des gains mesurables en termes de vitesse d'exécution.

Cependant, il est essentiel de noter certaines limitations de notre approche. Nous avons pris la décision stratégique de toujours laisser au moins quatre registres libres dans le programme. Ce choix, bien que arbitraire, découle de la reconnaissance que les expressions binaires sont parmi les opérations les plus consommatrices en termes de registres.

L'allocation de registres est un problème NP-complet[5], ce qui signifie qu'il devient impraticable de garantir une solution optimale dans un temps raisonnable, surtout pour des programmes complexes. Par conséquent, laisser un certain nombre de registres libres constitue une mesure pragmatique pour éviter de se perdre dans la complexité du problème d'allocation tout en préservant une marge de manœuvre pour des opérations cruciales.

Bien que cette décision ait été prise pour des raisons de praticité, elle entraîne une limitation de l'optimisation lorsque le nombre de registres disponibles est limité. L'efficacité de notre approche peut être compromise dans des scénarios où l'allocation des registres devient une ressource critique.

En fin de compte, le compromis entre laisser un nombre fixe de registres libres et optimiser au maximum dépend des spécificités du programme. Un équilibre délicat doit être maintenu pour éviter des opérations constantes de sauvegarde et de restauration vers la pile (push et pop), qui peuvent annuler

[5] https://www.geeksforgeeks.org/register-allocations-in-code-generation/

les gains potentiels d'optimisation.

En conclusion, notre algorithme d'optimisation, bien qu'efficace dans la réduction des accès à la mémoire dans les boucles "while", est associé à un compromis délibéré dans le nombre de registres laissés libres. Cette décision arbitraire souligne la nécessité de trouver un équilibre entre la complexité du problème d'allocation de registres et les avantages potentiels de l'optimisation dans des scénarios spécifiques.

2.4 Loop Inversion

La Loop Inversion[6], également connue sous le nom de Loop Interchange, apporte une optimisation significative en évitant les sauts conditionnels vers la fin de la boucle dans le code assembleur. Cela s'avère particulièrement bénéfique, en particulier dans le contexte de boucles imbriquées. Explorons davantage cette optimisation et ses avantages spécifiques.

L'une des caractéristiques clés de la Loop Inversion est la capacité à éliminer ou réduire considérablement les sauts conditionnels dans le code assembleur généré.

Dans le contexte de boucles imbriquées, la Loop Inversion peut encore être bénéfique en réduisant les sauts conditionnels associés aux sorties des boucles internes.

```
int function() {
    int x = 0;
    while (x < 10) {
        x = x + 1;
    }
}
```

Listing 2: Avant Loop Inversion

[6] https://en.wikipedia.org/wiki/Loop_inversion

```
   int function() {
                int x = 0;
                if (x<10) {
                        do {
                x = x +1;
                        } while ( x < 10 )
    }
}
```

Listing 3: Après Loop Inversion

2.4.1 Démonstration

Pour démontrer l'efficacité de l'optimisation nous allons fournir du pseudo-code assembleur:

```
    x := 0
L1: if x >= 10 goto L2
    x := x + 1
    goto L1
L2:
// in the 9th iteration
goto L1
if x >= 10
    x := x + 1
    goto L1
if i >= 10
    goto L2
```

Listing 4: Sans la Loop Inversion

```
  x := 0
      if x >= 10 goto L2
 L1:
      x := x + 1
      if x < 10 goto L1
 L2:
// in the 9th iteration
if x < 10
    goto L1
    x := x + 1
if i < 100
```

Listing 5: Avec la Loop Inversion

2.5 Dead Code Elimination

La phase d'élimination du code mort, également connue sous le nom de "Dead Code Elimination" (DCE), est une étape cruciale dans le processus de compilation visant à améliorer l'efficacité et la performance du code généré. Dans notre projet, nous avons intégré cette phase entre les passes de propagation de constantes et de repliement constant, reconnaissant son potentiel pour nettoyer le code source de sections inutiles, non atteintes ou redondantes.

Objectif de l'élimination du code mort :

L'objectif principal de l'élimination du code mort est de détecter et de supprimer les portions de code qui ne contribuent pas au résultat final du programme. Ces portions peuvent être des variables inutilisées, des branches de code inatteignables, ou des calculs dont les résultats ne sont jamais utilisés. En éliminant ces éléments superflus, nous réduisons la taille du programme généré et améliorons la lisibilité du code. La DCE consiste à

identifier les parties du code qui sont inatteignables à partir du point d'entrée du programme. Cela peut être réalisé en utilisant des analyses de flux de contrôle et de données pour déterminer quelles instructions ne seront jamais exécutées.

```
int function() {
        int a = 24;
                  int b = 24;
        // Unreachable code
        if (false) {
            return 25;
            a = 25;//dead code
        }
        return a;
```

Listing 6: Après Loop Inversion

```
int function() {
        int a = 24;
                  int b = 24;
        return a;
        }
```

Listing 7: Après Loop Inversion

3 Testes et Perf

3.1 Tests

3.1.1 Big Whiles, Big Dreams

La meilleure approche pour évaluer le bon fonctionnement d'un compilateur réside souvent dans la mise en place de tests ciblés, en particulier pour des structures clés telles que les boucles "while". Ces tests permettent de mesurer la performance du compilateur en termes de gestion des boucles, d'optimisations appliquées, et d'élimination de code mort. Pour illustrer cette démarche, nous fournissons un exemple de test ciblé sur les boucles "while".

```
{
    int x = 0;
    int y = 1;
    boolean b = false;

    while(x < 1.0e+6){
        x = x + 1;
        while(y < 1048576){
            y = y * 4;
        }
        b = !b;
    }


}
```

Listing 8: File: bigWhile1.deca

Nous avons écrit utiliser ce test plusieurs fois en changeant la taille de la boucle pour bien illustrer l'effet des algorithmes

3.1.2 I can't do math

Pour évaluer la robustesse de notre compilateur, nous avons conçu des tests spécifiques mettant en lumière la propagation, le constant folding, et l'intégration côté objet avec notre extension. L'exemple ci-dessous illustre ces aspects.

3.1.3 Script d'exécution

Nous avons mis en place un script de tests pour évaluer les performances de nos programmes dans les versions optimisée et non optimisée. En utilisant la commande time, le script mesure le temps d'exécution de chaque programme et enregistre les résultats dans un fichier perf.log. Cette approche

```
class GoodLife{
    int luckyNumber =  ((50 * 3) - (10 + 5) + (2 * 8) * 2) +
    ((9 / 3) * 111) - ((20 * 2) / 4) +
    ((15 * 4) + (60 / 2) - (5 * 2) * 2)
    - (9 / 3) + 1 + 219; // = 777

}

{
    int i = (1+11+1+1+1+1+1+1+11+1+1+1+11+1+1+1+1+11+1+1+1+1)*0;
    int j = i;
    int power = 1;
    GoodLife me = new GoodLife();

    while(i < 1000){
        i = i + 1;
        while(j < 10000000){
            j = j + 1;
            power = power * 4;
        }

        me = new GoodLife();
    }

}
```

Listing 9: File: iCantMath.deca

automatisée nous permet de surveiller les performances tout en nous offrant la flexibilité de vaquer à d'autres tâches pendant que les programmes s'exécutent. L'utilisation de time nous fournit des informations détaillées sur le temps écoulé, nous permettant ainsi de comparer efficacement les performances entre les différentes versions de nos programmes.

```
#!/bin/bash

cd "$(dirname "$0")"/../../.. || exit 1

source_directory=./src/test/deca/codegen/valid/unprovided/optimTests
destination_directory=./src/test/deca/codegen/valid/unprovided/generated
perf_directory=./src/test/deca/codegen/valid/unprovided/

timestamp=$(date +"%Y-%m-%d %HH%Mmin%Ss")
```

```
perf_file="$perf_directory/perf.log"

# Create the destination directory if it doesn't exist
mkdir -p "$destination_directory"

# Create the timestamped performance log file
echo "Timestamp: $timestamp" >> "$perf_file"

# Iterate through all .deca files and compile them
for decaFile in "$source_directory"/*.deca; do
    # Get the filename without extension
    filename=$(basename "$decaFile" .deca)

    # Compile the .deca file and move the .ass file to the destination directory
    decac "$decaFile"
    mv "$source_directory"/"$filename".ass "$destination_directory"

    decac -op "$decaFile"
    mv "$source_directory"/"$filename".ass
    "$destination_directory"/"$filename"op.ass
    echo "" >> "$perf_file"
    echo "$filename performances are :" >> "$perf_file"
    { time ima "$destination_directory"/"$filename".ass; } 2>&1 |
    grep real | cut -d' ' -f2 >> "$perf_file"

    echo "$filename optimized performances are :" >> "$perf_file"
    { time ima "$destination_directory"/"$filename"op.ass; } 2>&1 |
    grep real | cut -d' ' -f2 >> "$perf_file"

    echo "" >> "$perf_file"

done
```

3.1.4 Comparaison de temps

Les résultats obtenus à la suite de nos tests de performance se révèlent particulièrement significatifs, mettant en évidence une différence notable

Table 1: Compilation Time Comparison

Test Case	Time (Not Optimized)	Time (Optimized)
bigWhile1	0m2.512s	0m1.982s
bigWhile2	0m25.107s	0m19.225s
bigWhile3	4m2.704s	3m6.298s
bigWhile4	40m21.982s	30m59.394s
iCantMath	0m18.258s	0m13.316s

entre les versions optimisée et non optimisée de notre programme. Cette disparité devient particulièrement apparente dans le cas du programme bigWhile4.dec, caractérisé par une boucle conséquente. En effet, la version non optimisée nécessite un temps d'exécution considérable, atteignant 40 minutes. En revanche, grâce à l'optimisation, le temps d'exécution est réduit de manière significative à 30 minutes. Ces résultats soulignent l'impact significatif de l'optimisation, démontrant sa capacité à améliorer de manière substantielle les performances, surtout lorsqu'il s'agit de traiter des structures de programme complexes.

4 Conclusion

> Optimization is the art of finding the perfect balance between efficiency and elegance, turning complexity into simplicity and transforming challenges into opportunities for brilliance.

En résumé, notre exploration des performances a porté sur deux axes d'optimisation clés au sein de notre compilateur. Tout d'abord, nous avons intégré l'algorithme de réduction de la force, ciblant spécifiquement les opérations de multiplication et de division par des puissances de 2. Cette optimisation vise à remplacer ces opérations coûteuses par des opérations de décalage, réduisant ainsi la complexité algorithmique. En parallèle, l'introduction de

la propagation de constantes et du pliage constant a été réalisée. Ces algorithmes interviennent après l'étape d'analyse du code pour identifier et simplifier les expressions constantes, contribuant ainsi à améliorer l'efficacité du programme.

Les résultats de nos tests ont révélé une différence significative de performances, en particulier dans le cas des boucles complexes, comme illustré

par bigWhile4.dec. Cependant, il est important de reconnaître que ces optimisations présentent des limitations, notamment en termes de variations de performances en fonction de la structure du code.

En conclusion, l'optimisation demeure une stratégie puissante pour améliorer les performances, mais elle nécessite une évaluation minutieuse et une compréhension approfondie des limites et des avantages spécifiques à chaque contexte.ment bénéfiques.

5 Final Quote

> "Optimization is the art of finding the perfect balance between efficiency and elegance, turning complexity into simplicity and transforming challenges into opportunities for brilliance."