

Optimisation du code par GCC

Evan VOYLES, Amaury RODRIGUEZ, Stefan GAŁKIEWICZ

20 janvier 2022

1 Notions de compilation

Il y a une particulière puissance caché derrière les trois lettres gcc - signifiant à la fois la 'Gnu Compiler Collection' et la commande à lancer dans le terminal pour compiler un program C. Enfin pas que. Quand on appelle une commande de la forme

```
gcc -o hello hello_world.c
```

on lance effectivement une multitude de processus qui travaillent scrupuleusement en silence. Il s'agit de :

1. Préprocesser le fichier .c
2. Compiler les fichiers processés pour créer des fichier d'objet (.o)
3. Relier des fichiers d'objet dans une exécutable

Ca veut dire quoi, préprocesser un fichier .c? En effet, à chaque fois qu'on écrit `#include <stdio.h>` pour inclure un fichier d'entête, avant la compilation un outil dit le préprocesseur va remplacer une ligne d'include avec tout le contenu du fichier d'entête. Alors la directive préprocesseur `#include` consiste en une opération de copier et coller. Il y a plusieurs d'autres directives qui sont processées avant de compiler, par exemple le lecteur reconnaitra peut-etre les directives

```
#if, #ifdef, #ifndef, #else, #elif
```

qui sont employées pour la compilation conditionnel (par exemple, inclure un fichier spécifique pour Windows si le système est Windows) où bien pour éviter d'inclure le même fichier plusieurs fois :

```
#ifndef MONFICHIER_H
#define MONFICHIER_H

/ Contenu du fichier monfichier.h /

...

#endif
```

Après que nos fichiers sont préprocessés, ils sont prêts pour être compilés. La compilation traduit des fichiers en C à des instructions de machine en binaire, dit des fichiers d'objet qui portent l'extension `.o`. Pour expérimenter chez vous, on peut donner l'option `-c` à `gcc` pour arrêter après la compilation et créer des fichiers d'objet. La dernière étape s'agit de regrouper tout les fichiers d'objet pertinents et de les emballer dans une seule exécutable. C'est quoi la différence concrète entre un 'program' C et une 'bibliothèque'? Une bibliothèque c'est une collection des fonctions et leur définitions tandis ce que un program contient la fonction spécial `main`, qui sert comme une porte d'entrée de l'exécution d'un program.

Alors finalement l'outil `ld`, dit le 'linker' en anglais, relie tous les fichiers `.o` dans un executable. Son travail est compliqué mais fondamentale. Grosso modo, `ld` trouve exactement où elle sont définies les fonctions extérieures qu'on appelle dans un program. Par exemple, pour un program simple HelloWorld on va utiliser la fonction `printf` qui est défini d'ailleurs. Au moment de créer l'exécutable, `ld` va chercher la définition de `printf` dans la bibliothèque standard et mettre les instructions binaires dans l'exécutable. La magie de `gcc` c'est qu'il a fait tout cela pour vous en coulisse - un vrai emploi ingrat.

2 L'Optimisation

2.1 `-O0`

2.2 `-O1`

2.3 `-O2`

2.4 `-O3`

Your text goes here.

3 Options Spécifiques

Comme on a vu dans le dernier section, il y a de nombreuses options activé avec chaque niveau optimisation et il ne serait pas intéressant de vous dénombrer que fait chacun. Donc, on va choisir juste quelques unes à étudier qui sont utilisé souvent

3.1 Enlever le code superflu

Pour optimiser le code le compilateur peut enlever du code redondant, en effet le but de l'optimisation du code est de réduire la taille et augmenter la vitesse d'exécution du code. Une des premières optimisations qui paraît évident est d'enlever le code qui ne sert pas. Il s'agit principalement de deux types du code redondant - dead code (DC, code mort) et dead storage (DS, stockage mort). Tout segment du code qui ne s'exécute pas est dit mort.

3.1.1 -fdce, -ftree-dce

Étudions le segment du code suivant :

```
if (1 < 2) {
    printf("1 est plus petit que 2");
} else {
    printf("Ca marche plus les maths");
}
```

Ici le compilateur va remarquer qu'il y a un segment de code mort et en déduire que l'on peut optimiser. Comme nous le voyons dans le code ci-dessus, la condition `1 < 2` est toujours vérifiée donc la condition du `if` n'a pas besoin d'être testée et le `else` ne sera jamais exécuté. Vu que la ramification d'un programme peut notablement effectuer des ralentissements d'exécution, gcc peut enlever la branche `if` et effacer le code mort, sous condition que les options `-fdce` et `-ftree-dce` soient activées.

En fait, l'optimisation du code mort est si importante que ça que gcc enlève automatiquement dans certains cas :

```
#include <stdio.h>

int main() {
    if (1 < 2) {
        printf("1 est plus petit que 2");
    } else {
        printf("Ca marche plus les maths");
    }
}
```

```
.LC0:
    .string "1 est plus petit que 2"
main:
    push    rbp
    mov     rbp, rsp
    mov     edi, OFFSET FLAT:.LC0
    mov     eax, 0
    call    printf
    mov     eax, 0
    pop     rbp
    ret
```

FIGURE 1 – Instructions en assemblée générées pour x86-64 par gcc pour un program simple.

N'ayez pas peur de l'assemblée ! Comme nous le pouvons constater, les instructions pour la deuxième branche de l'expression `if` ne sont même pas générées (indiqué par le manque de couleur surlignante l'expression `printf`) !

On peut comparer cet exemple avec un program simple similaire, mais qui diffère cette fois-ci parce que le compilateur ne peut pas déterminer si la condition est toujours vérifiée. Étudions le program suivant.

```
// Type your code here, or load an example.
#include <stdio.h>

int main() {
    int x = 0;
    scanf("%d", &x);

    if (x < 100) {
        printf("x est inferieur a 100");
    } else {
        printf("x est superieur ou egale a 100");
    }
}

.LC0:
    .string "%d"
.LC1:
    .string "x est inferieur a 100"
.LC2:
    .string "x est superieur ou egale a 100"
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], 0
    lea     rax, [rbp-4]
    mov     rsi, rax
    mov     edi, OFFSET FLAT:.LC0
    mov     eax, 0
    call    __isoc99_scanf
    mov     eax, DWORD PTR [rbp-4]
    cmp     eax, 99
    jg      .L2
    mov     edi, OFFSET FLAT:.LC1
    mov     eax, 0
    call    printf
    jmp     .L3
.L2:
    mov     edi, OFFSET FLAT:.LC2
    mov     eax, 0
    call    printf
.L3:
    mov     eax, 0
    leave
    ret
```

FIGURE 2 – Exemple où des instructions sont générées pour les deux branches du if

Vu que la condition dans le if est dépendant sur une donnée d'entrée qui se processe à l'exécution, il est impossible de déterminer au moment de compilation si le code est mort. Du coup, il n'y a aucun optimisation dans cet exemple.

3.1.2 -fdse, -ftree-dse

Ces options traitent le cas où il y a des variables mortes - c'est-à-dire des variables qui ne sont jamais accédées. Par exemple dans le code qui suit, la variable y n'est pas utilisée. Pour optimiser le code il suffit juste de retirer la ligne et ainsi réduire la taille et augmenter la vitesse d'exécution du code. Cela marche aussi avec des fonctions qui ne font rien ou ne sont pas utilisés.

```
int my_func() {
    int x = 5;
    int y = 5;
    return x;
}
```

On peut étudier l'assemblé pour visualiser les optimisation fait par gcc.

| | | | |
|-------------|----------------------------|-------------|------------|
| my_func() : | | my_func() : | |
| | push rbp | | mov eax, 5 |
| | mov rbp, rsp | | ret |
| | mov DWORD PTR [rbp-4], 5 | | |
| | mov DWORD PTR [rbp-8], 5 | | |
| | mov eax, DWORD PTR [rbp-4] | | |
| | pop rbp | | |
| | ret | | |

(a) -O0

(b) -O1

FIGURE 3 – Exemple où le DSE est enlevé

La fonction entière est remplacée par une instruction de charger la valeur 5 dans un registre du CPU.