

Optimisation du code par GCC

Evan VOYLES, Amaury RODRIGUEZ, Stefan GAŁKIEWICZ

20 janvier 2022

Notions de compilation

Il y a une puissance particulière 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 exécutable. 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.

L'Optimisation

Pendant l'étape de compilation, le compilateur peut analyser votre code et effectuer des optimisations, si vous le souhaitez. Il y a une centaine des options pour le compilateur dont au moins cinquante pour l'optimisation. Les options pour changer le comportement d'optimisation commence par un `-f`, suivi par le nom de l'option. Comme il y a des nombreuses optimisations spécifiques, `gcc` a regroupé les stratégies d'optimisation qui ont le même objectif de compilation. C'est-à-dire que certaines optimisations pourraient rendre le code plus vite mais également augmenter la taille de l'exécutable. Alors les classes se divisent en différents niveaux de `-O`.

`-O0`

Désactiver la plupart des optimisations.

`-O1`

Le principe c'est de réduire le temps d'exécution aussi la taille du binaire. Au niveau `-O1` la compilation est un peu plus longue mais pas beaucoup comme à ce niveau les optimisations restent quand même simples.

Le premier niveau d'optimisation active les options suivantes

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fmove-loop-stores
```

```
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftime-bit-ccp
-ftime-ccp
-ftime-ch
-ftime-coalesce-vars
-ftime-copy-prop
-ftime-dce
-ftime-dominator-opts
-ftime-dse
-ftime-forwprop
-ftime-fre
-ftime-hiprop
-ftime-pta
-ftime-scev-cprop
-ftime-sink
-ftime-slsr
-ftime-sra
-ftime-ter
-funit-at-a-time
```

-O2

Le deuxième niveau active plus des optimisations. Sans échanger la taille de l'exécutable pour la vitesse du programme, ce niveau ralentit le temps de compilation mais augmente la performance des instructions générées. Les options activées à ce niveau-là ce sont toutes les options de -O2 plus :

```
-falign-functions -falign-jumps
-falign-labels -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-fnate-loops
-fgcse -fgcse-lm
-fhoist-adjacent-loads
-finline-functions
-finline-small-functions
-findirect-inlining
-fipa-bit-cp -fipa-cp -fipa-icf
-fipa-ra -fipa-sra -fipa-vrp
-fisolate-erroneous-paths-dereference
-fira-remat
```

```
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-loop-vectorize
-ftree-pre
-ftree-slp-vectorize
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp
-fvect-cost-model=very-cheap
```

-O3

Un troisième niveau active les optimisations les plus violentes et n'hésite pas à créer un exécutable plus grand au nom de la vitesse. Les options activées dans ce niveau sont :

```
-falign-functions -falign-jumps
-falign-labels -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-finite-loops
-fgcse -fgcse-lm
-fhoist-adjacent-loads
-finline-functions
-finline-small-functions
-findirect-inlining
-fipa-bit-cp -fipa-cp -fipa-icf
-fipa-ra -fipa-sra -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-loop-vectorize
-ftree-pre
-ftree-slp-vectorize
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp
-fvect-cost-model=very-cheap
```

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.

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.

`-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 vérifié donc la condition du `if` n'a pas besoin d'être testé et le `else` ne sera jamais exécuté. Vu que la ramification d'un programme peut notoirement 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 important que ca que gcc enlève automatiquement dans certains cas :

<pre>#include <stdio.h> int main() { if (1 < 2) { printf("1 est plus petit que 2"); } else { printf("Ca marche plus les maths"); } }</pre>	<pre>.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</pre>
--	---

FIGURE 4 – Instructions en assemblée générées pour x86-64 par gcc pour un program simple. Le code mort après else ne se traduit même pas.

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 la deuxième 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 vérifié. Étudions le program suivant.

<pre>// 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"); } }</pre>	<pre>.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</pre>
--	--

FIGURE 5 – Exemple où des instructions sont générées pour les deux branches du if

Vu que la condition dans le if est dependant sur une donnée d'entrée qui se pro-cesse a 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.

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

(a) -O0

```
my_func():
    mov     eax, 5
    ret
```

(b) -O1

FIGURE 6 – Exemple où le DSE est enlevé

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

Optimisation des boucles

Nous allons étudier ici quelques options qui permettent d'optimiser la compilation des boucles.

`-fpeel-loops`

Le peeling de boucle est un cas particulier du découpage de boucle. Le découpage de boucle est une méthode d'optimisation du compilateur. Celui-ci élimine les dépendances. Il simplifie la boucle en plusieurs boucles afin d'éliminer des dépendances. Ainsi ces boucles auront le même corps mais vont itérer différentes parties de la boucle. La particularité du peeling de boucle va être que si le compilateur détecte une ou quelques itérations problématiques il va la sortir de la boucle et l'exécuter en dehors de la boucle (c'est pour cela que ça s'appelle l'épluchage). Ainsi si l'on a par exemple

```
int p = 10;
for (int i=0; i<10; ++i) {
```

```
    y[i] = x[i] + x[p];
    p = i;
}
```

le compilateur avec fpeel-loops va remarquer que p=10 seulement a la première itération et donc il va compiler comme si le code était écrit

```
y[0] = x[0] + x[10];
for (int i=1; i<10; ++i) {
    y[i] = x[i] + x[i-1];
}
```

et donc on peut remarquer que le p ici n'est plus utilisé et est remplacé par directement 10 et la première itération est sorti du corps de la boucle et est remplacé par `y[0] = x[0] + x[10];`

-fmove-loop-invariants

Le loop invariant loop est une optimisation du compilateur qui permet de sortir un invariant d'une boucle sans changer sa sémantique. Par exemple :

```
int i = 0;
while (i < n) {
    x = y + z;
    a[i] = 6 * i + x * x;
    ++i;
}
```

On peut voir que `x=y+z` est un invariant dans la boucle et donc le `fmove-loop-invariants` va le sortir de la boucle sans en changer le sens.

-funswitch-loops

le funswitch-loops est une méthode d'optimisation du compilateur qui va permettre de sortir une condition située à l'intérieur d'une boucle à l'extérieur de celle-ci. Pour ceux elle va dupliquer le corps de la boucle en en plaçant à l'intérieur de conditions qui étaient de base à l'intérieur. Cela permet d'avoir une meilleure parallélisation de la boucle (c'est une technique d'optimisation de boucle) et donc améliore la vitesse du programme. Montrons un exemple de funswitch-loops

```
int i, w, x[1000], y[1000];
for (i = 0; i < 1000; i++) {
    x[i] += y[i];
    if (w)
        y[i] = 0;
}
```

ici on peut sortir le 'if (w)' et dupliquer les boucles for. Ainsi le compilateur pourra transformer cela en

```
int i, w, x[1000], y[1000];
  if (w) {
    for (i = 0; i < 1000; i++) {
      x[i] += y[i];
      y[i] = 0;
    }
  } else {
    for (i = 0; i < 1000; i++) {
      x[i] += y[i];
    }
  }
}
```

Dernières Remarques

Comme il y a un vaste nombre des options d'optimisations, on aurait aimé de vous en parler de plus ! Malheureusement on aura plus de l'espace. Ce qu'il faudrait retenir du coup pour la suite c'est qu'il y a des niveaux d'optimisations constructifs qui permet au programmeur de choisir si l'on veut réduire la taille du binaire ou le temps d'exécution. On peut activer une groupe des options à la fois avec les options -O1, -O2, ou -O3. Il y a aussi des autres groupes de O telles que -Og, -Oz, -Os, -Ofast que je vous encourage de vous en renseigner. On peut activer des optimisations spécifiques avec une option commençant par -f, -fno sinon pour le désactiver. Par exemple, l'on a -ffast-math pour activer ou bien -fnofast-math pour le désactiver. En conclusion, on espère que ce rapport vous a appris quelque chose du processus de compilation et qu'on vous a allumé une appréciation pour la magie qui est gcc.