PG06

Zeste de savoir

Attention aux optimisations de Clang !

15 janvier 2019

Table des matières

1. Introduction

> Si vous voulez en savoir plus sur les diverses optimisations réalisées par les compilateurs, allez consulter ce très bon tutoriel .

Bonjour !

J'écris ce billet (mon premier, soyez indulgents) pour vous montrer un petit exemple dans lequel les optimisations mises en place lors de la compilation peuvent créer des comportements inattendus, selon le programme utilisé. Dans le cas présenté ci-dessous, en langage C, le problème se produit avec **Clang** mais pas avec **GCC**.

2. Le code

Le code que nous allons utiliser va servir à chercher un contre-exemple de la conjecture de Syracuse . Pour rappel, la suite de Syracuse se construit de la façon suivante : on choisit un entier supérieur à 0, s'il est pair on le divise par 2 et s'il est impair on le multiplie par 3 et on ajoute 1. Puis on recommence avec la nouvelle valeur et ainsi de suite.

$u_0 = N \in \mathbb{N}^*$

$$\forall n \in \mathbb{N}\ ,\ u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ pair} \\ 3 \times u_n + 1 & \text{si } u_n \text{ impair} \end{cases}$$

Il semblerait que la suite de Syracuse se termine systématiquement par la boucle $4 > 2 > 1 > 4 > \ldots$ quel que soit le nombre de départ. Mais la conjecture n'a encore jamais été démontrée, il se pourrait donc qu'il existe des nombres pour lesquels la suite ne retombe jamais à 1.

Le code ci-dessous va chercher un éventuel contre-exemple en testant la suite de Syracuse pour chaque nombre. Tant qu'il n'a pas trouvé, il reste en boucle infinie. Seule la découverte d'un contre-exemple met fin au programme.

```c
// Cherche un contre-exemple de la conjecture de Syracuse
// dont le vol ne finit pas par 4 > 2 > 1 > 4 > ...

#include <stdio.h>

// Valeur maximum à tester (unsigned int)
// pour éviter l'overflow lors du 3*x+1
#define MAXVAL 1000000000

// Retourne 1 si un contre-exemple est trouvé, et boucle sinon
int syracuse(void)
{
	unsigned int nombre, vol, i;

	while(1) // Boucle infinie s'il n'y a pas de contre-exemple
	{
		for(nombre = 1; nombre <= MAXVAL; nombre++) // On teste chaque entier
		{
			vol = nombre;
			for(i = 0; i <= MAXVAL; i++)
			{
				if(vol == 1) break; // On retombe à 1, on passe à l'entier suivant
				if(vol >= MAXVAL) break; // Vol trop grand, on passe à l'entier suivant

				if(vol&1) vol = 3 * vol + 1; // Impair
				else vol = vol / 2; // Pair
			}

			// Si ici on a fait MAXVAL+1 étapes sans retomber à 1,
			// on a forcément fait une boucle à un moment, différente de 4 > 2 > 1 !
			if(i == MAXVAL + 1) return 1; // Contre-exemple trouvé
		}
	}
	return 0; // Code mort
}

int main(void)
{
	if(syracuse())
	{
		printf("Contre-exemple trouvé !\n");
```

```
42          }
43          return 0;
44  }
```

On est bien d'accord, ce programme est idiot, ne sert à rien, est mal foutu et occupe inutilement le CPU. Il n'y a aucune chance pour qu'il trouve un contre-exemple et il ne sert qu'à illustrer ce billet avec un cas plus ou moins concret.

3. Avec GCC

Bon, commençons par compiler notre ficher *syracuse.c* sans aucune optimisation avec **GCC**. La commande (avec Ubuntu) est la suivante :

```
1  $ gcc -O0 -o test_syracuse syracuse.c
```

On exécute alors le programme et…

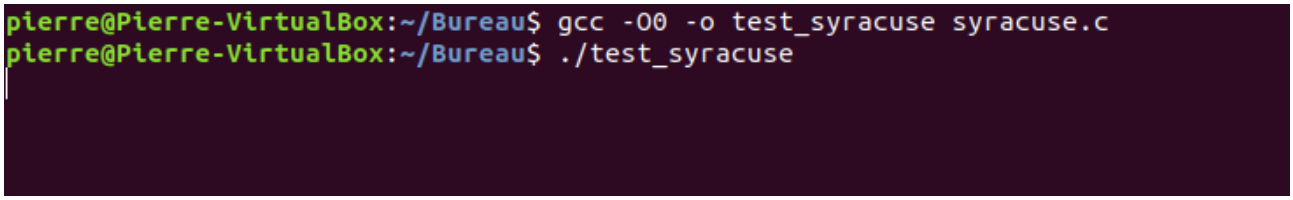

FIGURE 3. – Ça mouline…

Je vous laisse attendre chez vous un résultat aussi longtemps que vous voulez, si vous souhaitez mettre votre patience à l'épreuve, mais moi je vais tuer le processus.

On pouvait s'y attendre, puisque plein d'autres mathématiciens avant nous ont déjà essayé avec des programmes bien plus performants. Tentons à présent d'utiliser toutes les optimisations de GCC avec la commande suivante :

```
1  $ gcc -O3 -o test_syracuse syracuse.c
```

Vous pouvez essayer vous-mêmes ou me faire confiance, le comportement est exactement le même (à savoir : rien à l'écran). Du coup comment savoir si les optimisations sont bien présentes ? On peut regarder le nombre de lignes du code assembleur (fichier *syracuse.s*) généré par GCC avant la création de l'exécutable final.

```
$ gcc -O0 -S syracuse.c
$ wc -l syracuse.s
143 syracuse.s
$ gcc -O3 -S syracuse.c
$ wc -l syracuse.s
105 syracuse.s
```

On constate qu'il y a 143 lignes sans les optimisations et seulement 105 avec, donc le compilateur a bien fait son travail.

4. Avec Clang

Passons maintenant à **Clang** pour compiler notre ficher *syracuse.c*, toujours sans aucune optimisation pour commencer. La commande est la suivante :

```
$ clang -O0 -o test_syracuse syracuse.c
```

Quand on exécute le programme, ~~ça alors !~~ il ne se passe toujours rien.

```
pierre@Pierre-VirtualBox:~/Bureau$ clang -O0 -o test_syracuse syracuse.c
pierre@Pierre-VirtualBox:~/Bureau$ ./test_syracuse
```

FIGURE 4. – Ça mouline encore...

Je vous sens presque déçus. Allez, on ajoute à présent les optimisations de Clang et on regarde le résultat :

```
$ clang -O3 -o test_syracuse syracuse.c
```

```
pierre@Pierre-VirtualBox:~/Bureau$ clang -O3 -o test_syracuse syracuse.c
pierre@Pierre-VirtualBox:~/Bureau$ ./test_syracuse
Contre-exemple trouvé !
pierre@Pierre-VirtualBox:~/Bureau$
```

FIGURE 4. – On a trouvé ! O_o

Hein ?!? Le programme s'arrête instantanément en annonçant sa magnifique victoire, tuant une conjecture vieille de 90 ans !

Petit indice pour la suite : le problème vient de Clang, pas des maths.

Alors que s'est-il passé ? Pourquoi ce résultat surprenant ? Il faut analyser le code assembleur généré par Clang pour mieux comprendre. Je vous rassure, pas besoin de connaître l'assembleur en détail, on ne va s'occuper que de quelques instructions.

Pour être exact, Clang ne produit pas vraiment de l'assembleur. En fait, il va compiler le C vers un langage qui ressemble à de l'assembleur et qui va ensuite être pris en charge par la LLVM (*Low Level Virtual Machine*) pour devenir un exécutable. LLVM est ainsi une seule infrastructure indépendante qui regroupe différents compilateurs selon les langages : pour le C, elle s'associe donc avec Clang. Dans la suite du billet, ces détails ne sont pas très importants donc on fera comme si c'était simplement de l'assembleur, mais c'est pour ça que je ne vais pas comparer les tailles des fichiers entre GCC et Clang.
Plus d'informations sur Clang et sur la LLVM. Le logo de ce billet est d'ailleurs celui de LLVM.

La commande pour obtenir le fichier assembleur *syracuse.s* sans optimisation est la suivante :

```
$ clang -O0 -S syracuse.c
```

Le fichier fait 113 lignes, nous ne nous intéresserons qu'à la fonction *main* :

```
main:                                   # @main
# %bb.0:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $0, -4(%ebp)
        calll   syracuse
        cmpl    $0, %eax
        je      .LBB1_2
# %bb.1:
        leal    .L.str, %eax
        movl    %eax, (%esp)
        calll   printf
        movl    %eax, -8(%ebp)          # 4-byte Spill
.LBB1_2:
        xorl    %eax, %eax
        addl    $24, %esp
        popl    %ebp
        retl
```

Ainsi, sans optimisation, on a les actions suivantes :

— Le programme appelle bien la fonction *syracuse* à la ligne **7** (*calll*).
— Puis, ligne **8**, la valeur de retour est comparée à 0 (*cmpl* : compare).
— Si c'est le cas (mais ça n'arrive jamais) on saute directement de la ligne **9** à la ligne **15** (*je* : jump if equal) puis fin.
— Sinon, le code continue jusqu'à la ligne **13** qui appelle *printf* (*calll*) et affiche la phrase, puis fin.

C'est bien le comportement attendu. Comparons maintenant avec les optimisations :

```
$ clang -O3 -S syracuse.c
```

Le fichier fait cette fois 80 lignes, regardons à nouveau la fonction *main* :

```
main:                                   # @main
# %bb.0:
        subl    $12, %esp
        movl    $.Lstr, (%esp)
        calll    puts
        xorl    %eax, %eax
        addl    $12, %esp
        retl
```

C'est beaucoup plus court. Et remarquez-vous un problème ? **L'appel à la fonction *syracuse* a totalement disparu !** Et en plus, même *printf* est parti ! Ce code optimisé réalise uniquement un appel à la fonction *puts* (ligne **5**) et s'arrête.

Voilà donc pourquoi le programme optimisé avec Clang affiche immédiatement « Contre-exemple trouvé ! » et se termine.

5. Explications

La mise en place à priori étrange de cette optimisation à l'extrême s'explique par la structure particulière du code de la fonction *syracuse* : tout est de la faute de la boucle infinie. Elle est sans effet de bord (elle ne modifie rien en dehors de ses variables locales) et provoque ainsi un **comportement indéterminé** en C.

Chaque compilateur possède ses propres règles et méthodes d'optimisation, plus ou moins efficaces et justes. Et en particulier dans le cas d'un comportement indéterminé, chacun peut faire comme bon lui semble, au risque d'avoir des résultats inattendus comme ici.

GCC a décidé dans un cas comme celui-ci de ne rien faire et de laisser la boucle en place. Concernant Clang, en revanche, il a constaté une chose : quel que soit le temps passé dans la boucle, quels que soient les calculs effectués à l'intérieur, la fonction ne peut retourner qu'une seule et unique valeur : un 1 (ligne **31**). Du coup, son optimisation est simple et radicale : inutile d'appeler *syracuse*, la ligne **39** sera toujours vraie et il ne reste plus qu'à afficher directement le texte. Pas mal comme amélioration, non ?

Pour éviter ce problème et obliger Clang a tenir compte de l'évolution des variables au sein de la boucle, la solution est d'utiliser le mot-clé *volatile*. La ligne **13** devient alors `volatile unsigned int nombre, vol, i;`.

Et concernant le *printf* ? Cette fois, GCC et Clang ont tous les deux remplacé la fonction par un *puts*. En effet, la ligne **41** satisfait trois critères qui permettent cette optimisation :

— La chaîne de caractères est constante, elle n'affiche pas le contenu d'une variable.
— La chaîne de caractères se termine par un retour à la ligne (`'\n'`).
— On ne récupère pas la valeur retournée par le *printf* après l'affichage.

Dans ce cas, le résultat de *puts* est très exactement le même que celui de *printf*, mais ce dernier est moins rapide et efficace car il doit analyser chaque caractère de la chaîne pour vérifier s'il n'y a pas des formats à interpréter (par exemple `%d`, `%c`, `%f`…). Donc les compilateurs ont tout intérêt à utiliser *puts* à la place.

Voilà, c'est la fin de ce billet. Si un jour vous constatez des comportements bizarres dans vos programmes sans pouvoir l'expliquer par rapport au code C qui semble correct, pensez peut-être à `volatile`. Que ce soit pour des projets scolaires ou professionnels, il m'a déjà aidé plusieurs fois !