Compiladores e Intérpretes Práctica 2

3 - Resolución de condicionales

Introducción

Esta técnica se basa en que si tenemos un bucle anidado dentro del que **existe una condición que se cumple o no para todos los elementos del bucle anidado**, como es el caso siguiente:

```
int i, k;
static float a, x[N], y[N];
for(k=0; k<ITER; k++)
        for(i=0; i<N; i++)
                if(a*k==0.0) x[i] = 0;
                else y[i] = x[i]*y[i];
```

Entonces podemos resolver dicha condición fuera del bucle anidado y aplicar su resultado para todos los elementos del bucle anidado. A continuación se muestra un ejemplo:

```
int i, k;
static float a, x[N], y[N];
for(k=0; k<ITER; k++) {
        if(a*k==0) {
                for(i=0; i<N; i++)
                        x[i] = 0;
        }
        else {
                for(i=0; i<N; i++)
                        y[i] = x[i]*y[i];
         }
}
```

Propósito de la técnica

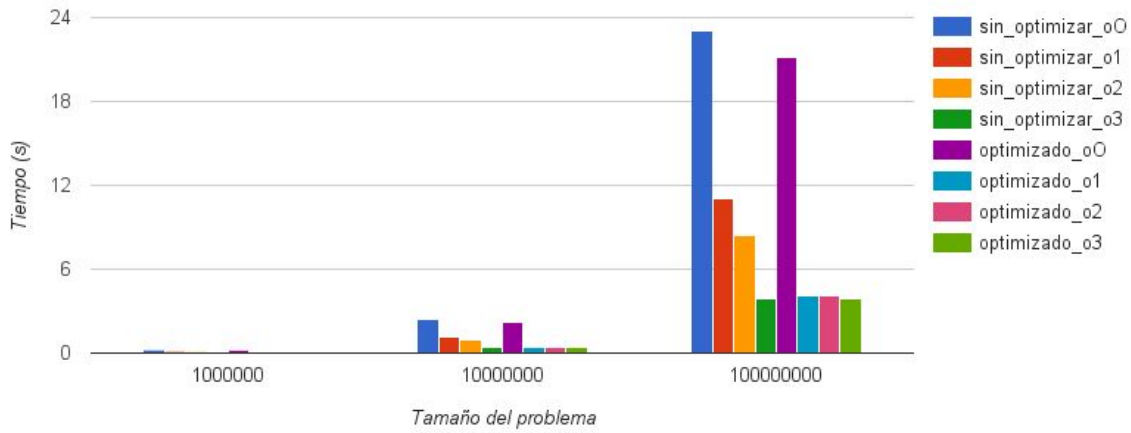
Al aplicar esta técnica, se persiguen las siguientes mejoras:

- **Reducir el número de ramificaciones que se producen.** En la versión sin optimizar se realizan N iteraciones del bucle y en cada una de ellas se realiza una comparación para decidir que rama del if se toma, pero en la versión optimizada solo se realiza una comparación fuera del bucle y a partir de ahí se realizan las N iteraciones del bucle, lo que obviamente supone un número mucho menor de comparaciones que se realizan.
- **Reducir el número de predicciones que realiza en tiempo de ejecución**. Al reducir el número de ramificaciones que se producen, también se está reduciendo el número de predicciones que se realizan en tiempo de ejecución..

Resultados obtenidos

A continuación, se muestran los resultados obtenidos de compilar el código anterior en su versión optimizada y sin optimizar, utilizando en ambos casos las diferentes opciones de optimización utilizadas por el compilador.

Para realizar la medición, en cuanto a tiempo de ejecución del código, se está midiendo todo el bloque de código mostrado anteriormente y se han añadido un par de instrucciones printf fuera del fragmento a medir para que el compilador no trate el código como código muerto.



	1000000	**10000000**	**100000000**
sin_optimizar_oO	0.2336890909	2.386045	23.10239155
sin_optimizar_o1	0.1118024545	1.160033636	11.09955045
sin_optimizar_o2	0.09084572727	0.9072323636	8.469395182
sin_optimizar_o3	0.01719718182	0.4020463636	3.868540455
optimizado_oO	0.2106871818	2.159469727	21.18505209
optimizado_o1	0.03808645455	0.4267117273	4.089688364
optimizado_o2	0.03626845455	0.4278717273	4.126239818
optimizado_o3	0.01729963636	0.3999737273	3.898795273

En este caso, podemos observar como las mejoras obtenidas al aplicar la optimización que estamos utilizando son de cerca del 10% en cuanto a tiempo de ejecución, independientemente del tamaño del problema, por los motivos explicados en el apartado anterior.

Así mismo, podemos observar como mientras las diferentes opciones de optimización aplicadas sobre la versión del código sin optimizar, mejoran en todos los casos el tiempo de ejecución. Cuando estas mismas se aplican sobre la versión optimizada, se obtienen con todas ellas casi el mismo resultado. En ambos casos las optimizaciones realizadas con el flag -O3 producen un código ensamblador bastante similar. De hecho, si lo comparamos mediante un comando diff podemos observar como casi solo difieren en el orden en el que realizan los if que hemos modificado.

```
1c1
<       .file   "main3.cpp"
---
>       .file   "mainOptimizado3.cpp"
26a27
>       jmp     .L5
29c30,38
< .L6:
---
> .L17:
>       xorl    %esi, %esi
>       movl    $400000000, %edx
>       movl    $_ZZ4mainE1x, %edi
>       addl    $1, %ebx
>       call    memset
>       cmpl    $100, %ebx
>       je      .L16
> .L5:
36,41c45,56
<       jne     .L8
<       movl    $400000000, %edx
<       xorl    %esi, %esi
<       movl    $_ZZ4mainE1x, %edi
<       call    memset
< .L4:
---
>       je      .L17
> .L8:
>       xorl    %eax, %eax
>       .p2align 4,,10
>       .p2align 3
> .L2:
>       movaps  _ZZ4mainE1x(%rax), %xmm0
>       addq    $16, %rax
>       mulps   _ZZ4mainE1y-16(%rax), %xmm0
>       movaps  %xmm0, _ZZ4mainE1y-16(%rax)
>       cmpq    $400000000, %rax
>       jne     .L2
```

```
44c59,60
<       jne     .L6
---
>       jne     .L5
> .L16:
67c83
<       jne     .L15
---
>       jne     .L18
74,76c90
<       .p2align 4,,10
<       .p2align 3
< .L8:
---
> .L18:
78,89d91
<       xorl    %eax, %eax
<       .p2align 4,,10
<       .p2align 3
< .L5:
<       movaps  _ZZ4mainE1x(%rax), %xmm0
<       addq    $16, %rax
<       mulps   _ZZ4mainE1y-16(%rax), %xmm0
<       movaps  %xmm0, _ZZ4mainE1y-16(%rax)
<       cmpq    $400000000, %rax
<       jne     .L5
<       jmp     .L4
< .L15:
```

Por último comentar, que no se ha visto que el tamaño del problema haya tenido algún tipo de influencia sobre la generación del código ensamblador.