Informe compiladores

Decembro 2020

Índice

1. Introdución

A finalidade desta práctica é analizar como funcionan algunhas técnicas de optimización do código, neste caso a substitución de operacións de multiplicacións e divisións enteiras por operacións de desprazamento. Para iso imos xerar dous programas coa mesma funcionalidade, un programa base sen optimización e outro ao que se lle aplicou a optimización.

A nosa intención é analizar a posible optimización e ver como afecta aos tempos de execución que se mediran coa función *gettimeofday*.

IMPORTANTE: A versión sen optimización tivo que ser cambiada xa que gcc facía a substitución de divisións e multiplicacións por desprazamentos aritméticos, parte fundamental do exercicio (**aínda coa opción -O0 activada**), de tal xeito que as medidas fosen resultasen pouco interesantes, xa que non nos permiten ver todo o potencial desta optimización.

2. Explicación da optimización

A optimización neste caso consiste na substitución das operacións e multiplicacións por un número potencia de 2 por desprazamentos de bits á dereita ou á esquerda.

O código do que partimos sen optimización é o seguinte:

```
1     for(j=0; j<ITER; j++){
2     //Medida interesante: Canto tarda de media en calcular o bucle N
3         for(i=0; i<N; i++) {
4           a = i * 8;//Multiplicacion e division directas
5           b += a / 32;
6         }
7     }
8
```

Figura 1: Bucle principal do programa sen optimización

Para intentar mellorar o rendemento imos facer dous cambios. En primeiro lugar imos quitar a dependencia que ten b coa variable a. Para iso quitamos a do bucle interno que vai desde 0 a N-1 e asignándolle o seu valor final, esto é $a = (N-1)/32$. Unha vez temos o valor definitivo de a podemos mirar como recalcular b sen ela.

$$b = \sum_{i=0}^{N-1} i * 8/32 \rightarrow \sum_{i=0}^{N-1} i/4$$

Con este cambio o código pasaría a ser este:

```
    for(j=0; j<ITER; j++){
    //Medida interesante: Canto tarda de media en calcular o bucle N
        for(i=0; i<N; i++) {
          b += i / 4;
        }
        a = (N-1) * 8;
    }

```

Figura 2: Bucle principal do programa cambiando a orde das operacións

O segundo cambio que debemos facer é substituír as operacións de multiplicación e divisións por desprazamentos de bits á esquerda e á dereita respectivamente.

```
    for(j=0; j<ITER; j++) {
          for(i=0; i<N; i++) {
            b += (i >> 2);
          }
            a= (N-1) << 3;
    }

```

Figura 3: Bucle principal do programa sen optimización

Estó só é posible porque estamos multiplicando e dividindo por potencias de dous. En caso de non ser así habería que incluír outras operacións de suma para poder realizar este prociso. Por exemplo se en vez de multiplicar por oito multiplicaramos por 6 ou por 3 teríamos que facer o seguinte

$$a = x * 6 \rightarrow a = (x << 2) + (x << 1)$$

$$a = x * 3 \rightarrow a = (x << 1) + x$$

A intención con esta substitución é cambiar a operación de multiplicación por unha de desprazamento que ten un menor custo computacional.

3. Execución das versións

Á hora de executar as imos medir o tempo medio que tarda en realizar **N iteracións**. Para iso medimos o tempo que dividimos o tempo que tarda en facer un número determinado de iteracións no bucle exterior e despois facer o tempo medio que tardou. O programa lánzase varias veces cun script en bash para tomar varias medidas e comparar mellor.

Vanse tomar medidas de tempo cos seguintes valores de N e ITER.

Valor de N	Valor de ITER
10	100000000
50	50000000
100	10000000
250	9000000
500	4000000
1000	900000
2500	900000
5000	400000
10000	100000
25000	100000
50000	100000
100000	10000
500000	10000
1000000	10000

Figura 4: Táboa de valores das constantes N e ITER para ambos experimentos

3.1. Execución da versión sen optimización

Comezamos coa versión sen optimización. Como se indicou arriba, **tívose que cambiar os números polos que se fai o produto e a división**, que pasaran a ser 7 e 31 en vez de 8 e 32. Esto tivo que facerse porque gcc estaba convertindo de xeito automático estas operacións por desprazamentos de bits.

Comezaremos vendo a parte do código en que se realiza a operación de multiplicación $a = i * 7$ que se corresponde co código ensamblador de abaixo:

```
.L7:
  movl  -80(%rbp), %edx
  movl  %edx, %eax
  sall  $3, %eax
  subl  %edx, %eax
  movl  %eax, -72(%rbp)
```

O que se fai en primeiro lugar é cargar o valor actual de i que se atopa na posición -80 do rexistro punteiro *%rbp* no punteiro **edx**. Cargamos este valor tamén no punteiro **eax** sobre o que executamos un desprazamento aritmético cada a esquerda de tres posicións *sall $3, %eax*. Unha vez feito o desprazamento réstase a *%eax* o contido de *%edx* que será i de tal xeito que $\%eax = i * 7$. A parte da multiplicación remata almacenando o contido de **%eax en -72 %rbp** onde se almacena o valor da variable a.

A continuación temos a operación de división entre 31. Para iso comezamos cargando de novo o contido da variable **a** (-72(%rbp)) en %eax. Unha vez feito iso pásase ese contido ao rexistro **%rdx**.

```
movl   %eax, -72(%rbp)
movl   -72(%rbp), %eax
movslq   %eax, %rdx
```

Tras realizar algúns cálculos intermedios chegamos ao cálculo do valor de b. Para iso comezamos sumando ao contido de edx, que sigue a ser i, o contido de eax ($\%eax = i * 7$). Deste xeito quédanos que $\%edx = i * 8$.

```
addl   %eax, %edx
sarl   $4, %edx
sarl   $31, %eax
subl   %eax, %edx
movl   %edx, %eax
addl   %eax, -68(%rbp)
addl   $1, -80(%rbp)
```

O seguinte paso é facer dous desprazamentos cara a dereita. O primeiro de catro posicións para dividir o valor de edx e o segundo de 31 posicións para dividir o valor de eax. Por último restamoslle ao valor de de edx o de eax, almacenamos o valor na posición da variable b e incrementamos o valor de i.

Toda esta sección repítese para N*ITER iteracións.

3.2. Execución da versión con optimización

A diferenza das probas realizadas sobre a versión sen optimización, estas mantéñense fieis ao exercicio orixinal. Xa se explicou na introdución os cambios introducidos, de xeito que aquí só explicaremos as seccións máis relevantes do programa en ensamblador.

Imos comezar co interior dos dous bucles: a operación $b+ = (i >> 2)$.

```
.L7:
  movl   -80(%rbp), %eax
  sarl   $2, %eax
  addl   %eax, -68(% rbp)
  addl   $1, -80(% rbp)
```

Como na sección anterior, vemos que se comeza cargando o valor de i no rexistro **eax**. Porén, a continuación realízase unha operación de desprazamento aritmético cara a dereita de dúas posicións, o que é equivalente a multiplicar dito elemento por catro. Actualizamos o contido de b que está almacenado na posición -68 de **rbp** e incrementamos o valor de i.

A diferenza do caso anterior, a segunda operación atópase fóra do bucle N. Neste caso, vemos no ensamblador que directamente se introduce o valor no rexistro que corresponde á variable a, este é a posición **-72 de rbp** e vemos que se actualiza xusto ao saír do bloque en que se realiza a multiplicación coa operación *movl $7999992, -72(%rbp)*.

```
.L6:
  cmpl   $999999, -80(% rbp)
```

```
3   jle .L7                         ;salto se i < N
4   movl  $7999992, -72(% rbp)      ;resultado da division
5   addl  $1, -76(% rbp)
```

Toda esta sección repítese para N*ITER iteracións.

4. Análise dos resultados

A optimización dada mellora á versión sen optimización para tamaños de N pequenos como se pode ver na gráfica. Porén, a medida que vai medrando N vanse aproximando cada vez máis o tempo entre ambas, ata chegar a $N = 5000$ no que se igualan. A partir de deste valor, a suposta **optimización da peores resultados que a versión base**.

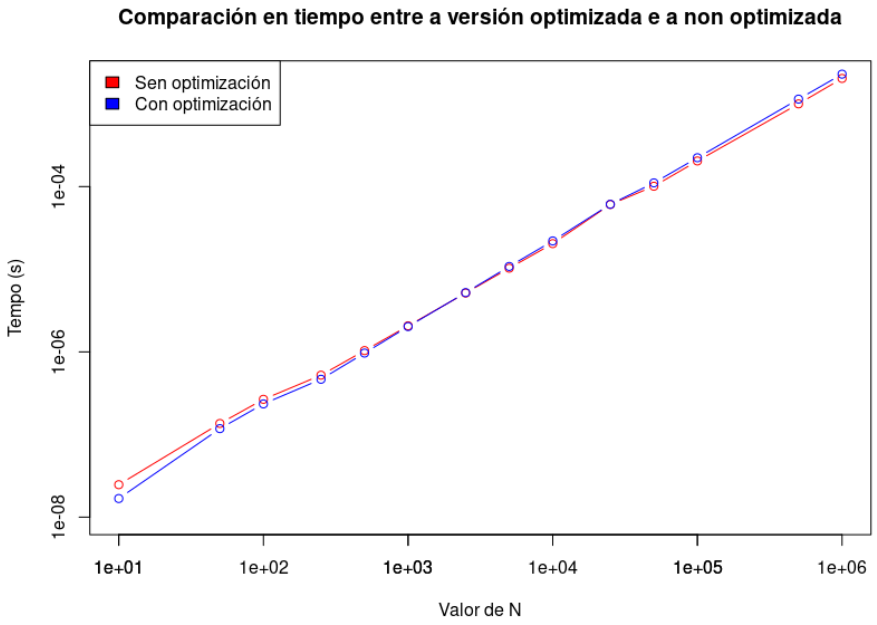

Figura 5: Gráfica dos resultados da experimentación.

Nesta gráfica vemos como se van comportando ambas opcións, seguindo o que se espuxo arriba como o rendemento da versión optimizada contra a versión base vai empeorando. É importante ter en conta á hora de visualizar os datos que a gráfica emprega unha escala logarítmica en ambos eixos, de modo que a distancia entre os puntos

A mellora obtida pola optimización para os tamaños de N pequenos non é demasiado significativa xa que cando empezan a chegar a tempos significativos ten mellores resultados a versión base. Esto

pódese apreciar mellor na gráfica de abaixo en que só está en escala logarítmica o eixo x.

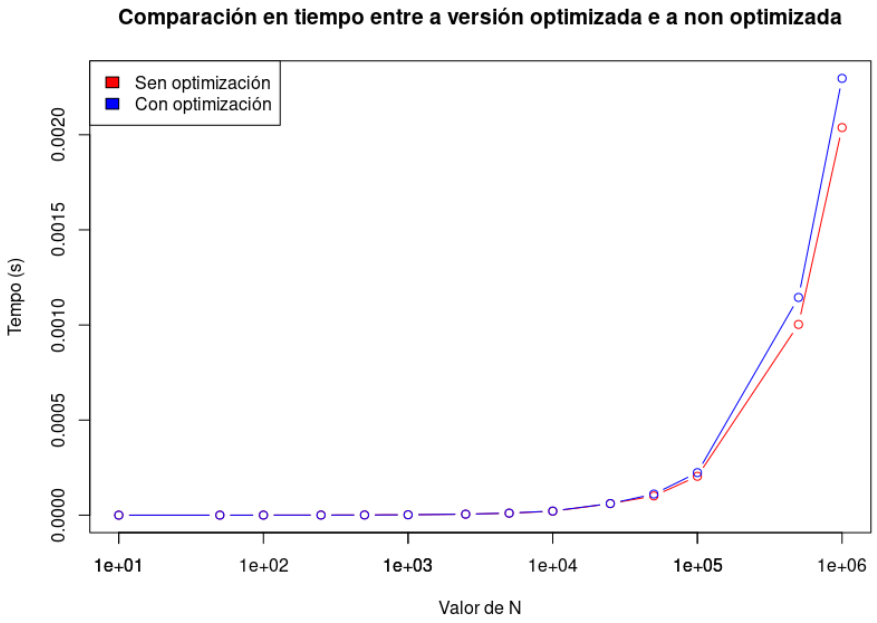

Figura 6: Gráfica onde se aprecia mellor a diferenza nos tempos entre ambas.

5. Conclusión

Pese a que os resultados estean algo influenciados pola substitución automática de divisións e multiplicacións enteiras por desprazamentos cara a esquerda que fai gcc ao traducir o código en C a x86-64m vemos que esta optimización si que funciona mellor que a versión base para valores de N máis pequenos, empezando a perder rendemento a medida que medra.

Pese a todo, como xa o propio *gcc* substituiu as operacións que se indicaron antes, non se puido ver con total certeza cal é o grado de mellora que se acadaría con dita optimización, xa que, ao tela ambas versións do executable, a diferenza real entre ambas versións apenas é relevante. Dado esto **os resultados obtidos non son concluíntes** de se se mellorou ou non o rendemento da versión base.