Compiladores e Intérpretes Práctica 2

2 - Considera la optimización de desenrolle de lazos internos

Introducción

Como el nombre de esta técnica indica, se basa en el desenrollar los bucles anidados para mejorar la velocidad de ejecución del código. Si la aplicamos sobre un código como el siguiente;

```
int i, k;
float a, b;
static float x[N], y[N];

for(k=0; k<ITER; k++)
        for(i=0; i<N; i++)
                y[i] = a * x[i] + b;
```

Tras aplicar la técnica, se podría generar un código como el que se muestra a continuación

```
int i, k;
float a, b;
static float x[N], y[N];

for(k=0; k<ITER; k++)
        for(i=0; i<N; i+=4){
                y[i] = a * x[i] + b;
                y[i+1] = a * x[i+1] + b;
                y[i+2] = a * x[i+2] + b;
                y[i+3] = a * x[i+3] + b;
        }
```

Propósito de la técnica

Al aplicar esta técnica, se persiguen las siguientes mejoras:

- **Aumentar el tamaño del bloque básico**, intentando así aumentar las posibilidades de que el compilador pueda aplicar técnicas de optimización adicionales por su cuenta. Esto se debe a que estamos aumentando el tamaño del cuerpo del bucle anidado al desenrollarlo.
- **Reducir el número de comparaciones que se realizan en el bucle anidado**. Tal cual está definido el bucle a optimizar, se realizan **N comparaciones por cada ITER**, al desenrollar el bucle, se procede a realizar solo **¼ de las comparaciones** anteriores.
- **Reducir el número de predicciones de salto que se realizan al alcanzar el final del bucle**. Al reducir el número de iteraciones del bucle interno, se alcanza menos

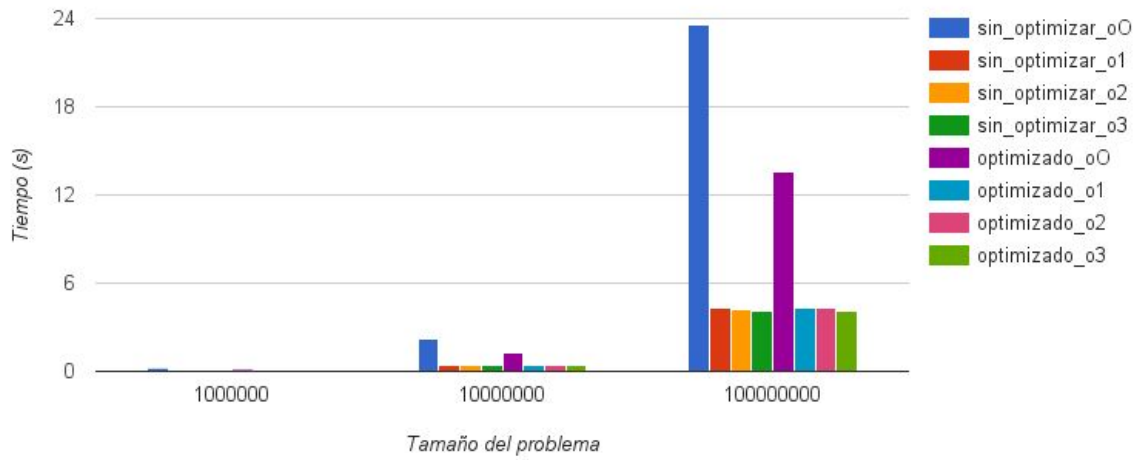
veces el final de este y por la tanto se realizan menos predicciones de salto durante la ejecución del programa.

- **Aumentar el paralelismo a nivel de instrucción**. Al aumentar el número de operaciones que se realizan por cada iteración del bucle, podemos aumentar el paralelismo a nivel de instrucción en procesadores superescalares y sobre todo en un caso tan sencillo como el que estamos manejando.

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	0.2223	2.2141	23.6150
sin_optimizar_o1	0.0366	0.4301	4.3166
sin_optimizar_o2	0.0358	0.4070	4.1711
sin_optimizar_o3	0.0165	0.3910	4.0549
optimizado_oO	0.1332	1.3151	13.6168
optimizado_o1	0.0361	0.4116	4.2794
optimizado_o2	0.0352	0.4149	4.2773
optimizado_o3	0.0166	0.3970	4.1116

Como se puede ver en los resultados obtenidos, la versión optimizada es cerca del doble de rápida respecto a la versión sin optimizar en la mayoría de los casos.

También es interesante destacar que el resto de flags de compilación utilizados han mejorado. prácticamente en la misma medida, ambas versiones del programa, existiendo unas diferencias de tiempo mínimas entre las diferentes opciones de compilación. Lo que implica que las optimizaciones que han sido muy efectivas en cuanto a tiempo de ejecución, han sido las aplicadas por el flag -O1, ya que el resto de flags apenas mejoran los resultados obtenidos por -O1

Así mismo, no se ha podido apreciar que en la versión del código optimizado, se haya realizado alguna optimización que no se realizase sobre la versión sin optimizar. Esto se ha comprobado realizando la operación diff sobre los código ensambladores de las versiones sin optimizar y las versiones optimizadas con el flag -O1.

diferencias entre las versiones sin optimizar

```
40c40,70
<       addl    $1, -72(%rbp)
---
>       movl    -72(%rbp), %eax
>       leal    1(%rax), %edx
>       movl    -72(%rbp), %eax
>       addl    $1, %eax
>       cltq
>       movss   _ZZ4mainE1x(,%rax,4), %xmm0
>       mulss   -64(%rbp), %xmm0
>       addss   -60(%rbp), %xmm0
>       movslq  %edx, %rax
>       movss   %xmm0, _ZZ4mainE1y(,%rax,4)
>       movl    -72(%rbp), %eax
>       leal    2(%rax), %edx
>       movl    -72(%rbp), %eax
>       addl    $2, %eax
>       cltq
>       movss   _ZZ4mainE1x(,%rax,4), %xmm0
>       mulss   -64(%rbp), %xmm0
>       addss   -60(%rbp), %xmm0
>       movslq  %edx, %rax
>       movss   %xmm0, _ZZ4mainE1y(,%rax,4)
>       movl    -72(%rbp), %eax
>       leal    3(%rax), %edx
>       movl    -72(%rbp), %eax
>       addl    $3, %eax
>       cltq
```

```
>       movss   _ZZ4mainE1x(,%rax,4), %xmm0
>       mulss   -64(%rbp), %xmm0
>       addss   -60(%rbp), %xmm0
>       movslq  %edx, %rax
>       movss   %xmm0, _ZZ4mainE1y(,%rax,4)
>       addl    $4, -72(%rbp)
```

diferencias entre las versiones optimizadas con -O1

```
28c28,31
<       addq    $4, %rax
---
>       movss   %xmm0, 4(%rax)
>       movss   %xmm0, 8(%rax)
>       movss   %xmm0, 12(%rax)
>       addq    $16, %rax
```

Como podemos observar en las comparaciones realizadas, la única optimización que no se ha aplicado exactamente sobre ambas versiones del código, es la substitución de las múltiples operaciones que se utilizan en el bucle desenrollado por una única operación coorespondiente a cada iteración desenrollada.

Por último, mencionar que no se aprecia ningún tipo de influencia del tamaño del problema sobre el código.ensamblador generado utilizando cada uno de los flags de optimización sobre ambas versiones del programa.