Práctica 3:

OPTIMIZACIÓN DE CÓDIGO

5º curso Ing. Informática
Arquitectura de Computadores I
José Antonio Guerrero Avilés
75485683M
cany@correo.ugr.es

Ejercicio 1 - Para el programa ejemplo1.c que se muestra en la Figura 1.

A. Modifique el código C para reducir el tiempo de ejecución del mismo. Justifique los resultados obtenidos a partir de la modificación realizada.

Primero he obtenido el tiempo que tarda el código original con una batería de pruebas que realiza lo siguiente:

- Ejecuta 50 veces el programa.
- Varía M con 50000, 100000 y 200000.
- Se calcula la media de los datos que se obtienen.

Este es el código del programa modificado para poder realizar la batería de pruebas:

```
#include <stdio.h>
#include <math.h>
#include <time.h>

struct {
      int a;
      int b;
}

s[500];
long ii,i;

main(int argc, char ** argv){

      clock_t start,stop;
      start= clock();

      for (ii=1;ii<=M;ii++){

            for (i=0;i<500;i++)
                  s[i].a=2*s[i].a;

            for (i=0;i<500;i++)
                  s[i].b=3*s[i].b;
      }

      stop = clock(); /* Final de medida de tiempo*/
      printf("%f \n",difftime(stop,start)/CLOCKS_PER_SEC);

      return 0;
}
```

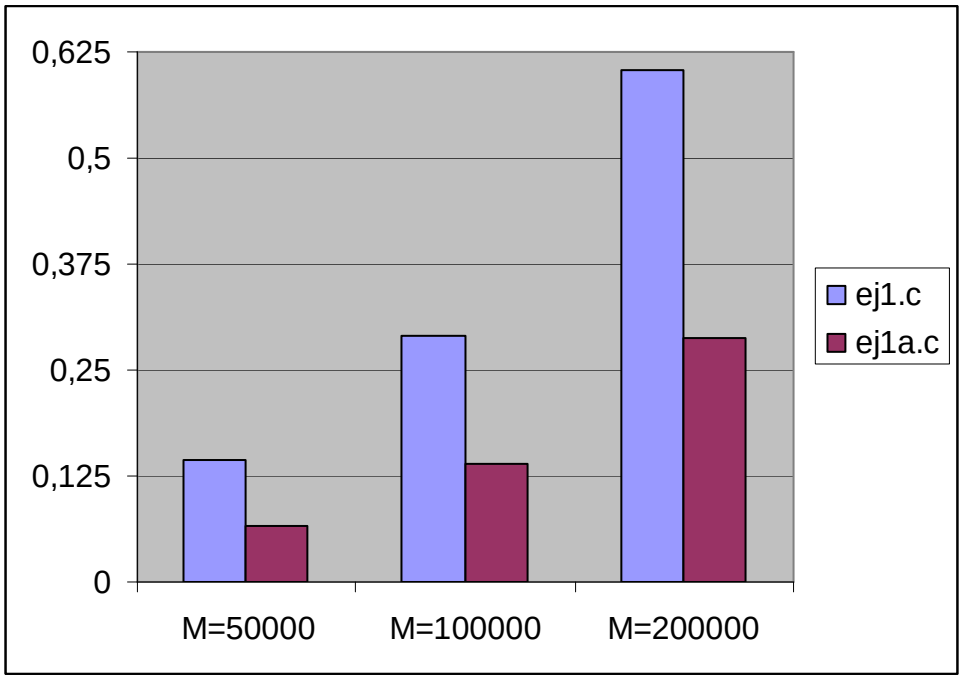
Para optimizar el código en C he fusionado los dos bucles for anidados, eliminando un bucle. Además, como la estructura de datos son 500 elementos, cada uno de ellos con un a y un b, en memoria tendría la forma abababababab... por lo que el acceso a los datos que se hacía antes (primero todos los a, y después todos los b) es menos eficiente que si accedes a los datos por orden, tal y como se hace con esta mejora.
También se ha desenrollado el bucle.

Este es el código del programa optimizado:

```
#include <stdio.h>
#include <math.h>
#include <time.h>
struct {
	int a;
	int b;
}
s[500];
long ii,i;

main(int argc, char ** argv) {

	clock_t start,stop;
	start= clock();

	for (ii=1;ii<=M;ii++){

		for (i=0;i<500;i++){

			s[i].a=2*s[i].a;
			s[i].b=3*s[i].b;
			s[i+1].a=2*s[i+1].a;
			s[i+1].b=3*s[i+1].b;
			s[i+2].a=2*s[i+2].a;
			s[i+2].b=3*s[i+2].b;
			s[i+3].a=2*s[i+3].a;
			s[i+3].b=3*s[i+3].b;
			s[i+4].a=2*s[i+4].a;
			s[i+4].b=3*s[i+4].b;
		}
	}

	stop = clock();
	printf("%f \n",difftime(stop,start)/CLOCKS_PER_SEC);

	return 0;
}
```

Tras ejecutar ambos, estos son los resultados:

Programa	M=50000	M=100000	M=200000
ej1.c	0,144	0,289	0,6026
ej1a.c	0,065	0,139	0,2879



B. Genere los programas en ensamblador para la implementación más eficiente de ejemplol.c del apartado anterior, considerando las distintas opciones de optimización del compilador (-O1, -O2,..) y explique los resultados obtenidos a partir de las características de dichos códigos.

- Al aplicar la opción –O1 se eliminan saltos innecesarios, consiguiendo reducir el número de instrucciones de salto y de etiquetas. Esto es debido a la opción fthreadjumps, que se encarga de comprobar si hay un salto a otra posición donde se encuentra otro salto y si se conoce la condición de salto se redirecciona el primero convenientemente. En realidad, en la versión actual del compilador, cuando realizamos la opción -O1 el flag -fthread-jumps está activado, como se puede ver si realizamos el comando gcc -c -Q -O1 --help=optimizers. Además se activan distintas opciones que antes de la optimización no estaban activas.

- Con la opción –O2 se activan algunas cosas respecto a –O1. Como podemos observar en el código se realizan alineamientos (con O1 no se había realizado ninguno (en las líneas 6, 18, 19,etc.). Otro cambio que se observa es reordenación de parte de código las líneas de la 47 a 52 con respecto de la 44 a 49 obtenidas en –O1. Además se sustituyen algunas instrucciones por alternativas más rápidas como las instrucciones de tipo movl $0, %eax por xorl %eax, %eax.

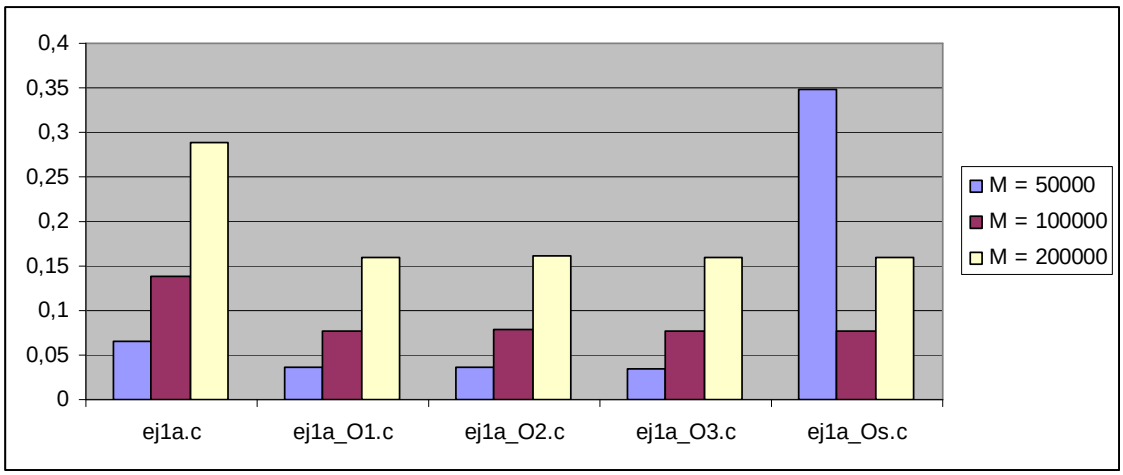
- Al realizar la opción de optimización -O3 no se produce ningún cambio respecto de la opción -O2.

- Al realizar la opción de optimización -Os se desactivan algunas opciones activadas en -O2 y se eliminan los alineamientos introducidos con la opción –O2 y el código no se reduce tanto como en las otras optimizaciones.

Una vez obtenidos los códigos, ejecutamos y obtenemos los siguientes datos:

	ej1a.c	**ej1a_O1.c**	**ej1a_O2.c**	**ej1a_O3.c**	**ej1a_Os.c**
M = 50000	0,065	0,0358	0,0356	0,035	0,348
M = 100000	0,139	0,0764	0,079	0,0774	0,076
M = 200000	0,2879	0,1596	0,1622	0,16	0,1594

En el siguiente gráfico, se puede observar mucho mejor los datos de la tabla anterior, y se ve como las optimizaciones mejoran el tiempo de ejecución:



C. Partiendo del código ensamblado del apartado anterior que justifique como más eficiente, realice modificaciones destinadas a mejorar sus prestaciones de tiempo. En las referencias encontrará recomendaciones sobre mejoras en el código ensamblador, independientes y aplicables a procesadores específicos.

Ejercicio 2 - El benchmark Linpack ha sido uno de los programas más ampliamente utilizados para evaluar las prestaciones de los computadores. De hecho, se utiliza como base en la lista de los 500 computadores más rápidos del mundo (el Top500 Report). El núcleo de este programa es una rutina denominada daxpy que multiplica un vector por una constante y los suma a otro vector:

for (i=1;i<=N,i++) y[i]=y[i]+a*x[i];

A. Siguiendo las indicaciones de diseño para la medición de prestaciones descritas en el Apéndice 3.2. Desarrolle en C un programa basado en el anterior bucle. Genere los programas en ensamblador para cada una de las opciones de optimización del compilador (-O1, -O2,..) y explique las diferencias que se observan en el código justificando las mejoras en velocidad que acarrean.

Este es el programa en c que he diseñado (archivo ej2a.c):

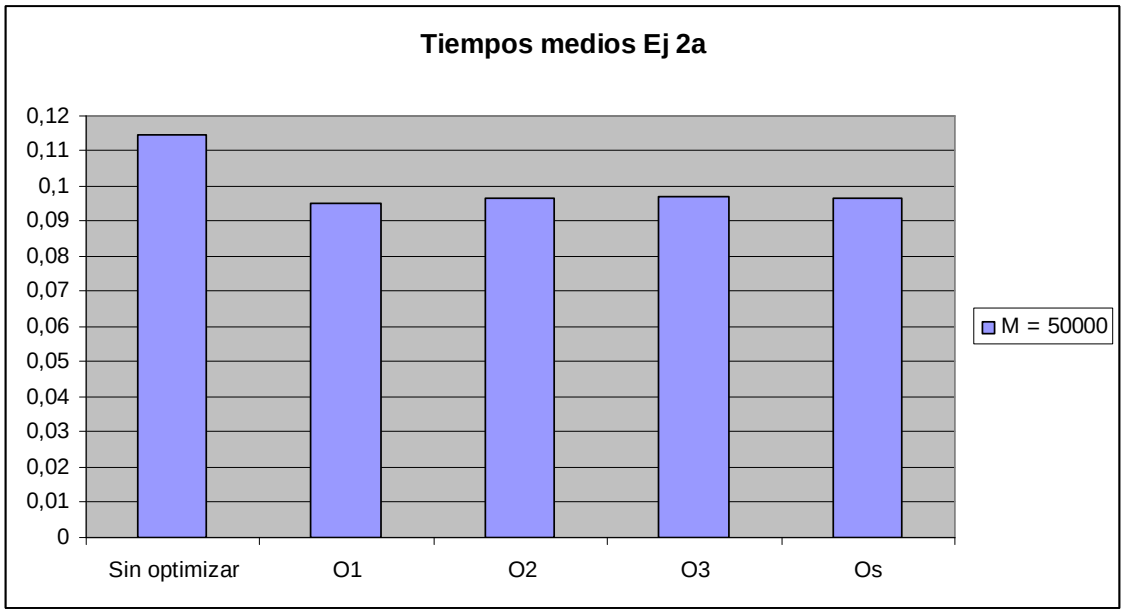
```
#include <stdio.h>
#include <math.h>
#include <time.h>

#define N 50000000
float y[N];
float x[N];
long i;

main() {

	clock_t start,stop;
	start= clock();

	for (i=1;i<=N;i++){

		y[i]=y[i]+5.0*x[i];
	 }


	 stop = clock();
	 printf("%f \n",difftime(stop,start)/CLOCKS_PER_SEC);
	 return 0;
}
```

Tras obtener los códigos en ensamblador con las distintas opciones de optimización, podemos ver que con este programa volvemos a apreciar cambios similares a los producidos y ya explicados en el ejercicio anterior, es decir, se reducen los saltos en O1 al suprimir los saltos obligados indicados por la instrucción jmp, además hay una reducción del número de instrucciones. En O2 se alinea el código al principio del programa y al inicio del bucle. Las mejoras introducidas en O3 no afectan a este programa por lo que queda igual que O2. Por último, en Os se desactivan funciones de optimización que provocan que no haya alineamiento y no se reduce el número de saltos.

A continuación, se muestran los tiempos medios (obtenidos de 50 ejecuciones para cada una de las optimizaciones), teniendo en cuenta que N se ha configurado como N= 5 millones.

	Sin optimizar	**O1**	**O2**	**O3**	**Os**
M = 5.000.000	0,1144	0,0952	0,0966	0,097	0,0966

En el siguiente gráfico se puede apreciar mucho mejor la diferencia entre los tiempos de las diferentes optimizaciones:



B. Para la mejor de las opciones, obtenga los tiempos de ejecución con distintos valores de N y determine para su sistema los valores de Rmax (valor máximo del número de operaciones en coma flotante por unidad de tiempo), Nmax (valor de N para el que se consigue Rmax), y N1/2 (valor de N para el que se obtiene Rmax/2). Estime el valor de la velocidad pico (Rpico) de su procesador y compárela con el valor obtenido para Rmax.

Tanto en el gráfico, como en la tabla anterior, se puede apreciar claramente que la mejor opción de optimización es la opción –O1.
Una vez visto esto, obtenemos la siguiente tabla con los tiempos de ejecución para distintos valores de N:

N	Tiempo	R
5.000.000	0,0986	101.419.878
10.000.000	0,2024	98.814.229,2
20.000.000	0,416	96.153.846,2
50.000.000	1,0586	94.464.386,9
100.000.000	2,1234	94.188.565,5

Donde R se ha obtenido como resultado de aplicar la siguiente fórmula:

R= NumeroOperacionesFP / Tiempo = 2 * N / Tiempo

Por tanto, observando la tabla, se ve claramente que:

- **Rmax** = 101.419.878
- **Nmax** =5.000.000

Ahora vamos a calcular Rmax/2 para poder saber cuál es N1/2:

Rmax/2 = 50.709.939

En la tabla podemos observar como R va disminuyendo conforme aumenta N.
De esta forma, y debido a que mi PC no me permite ejecutar el programa con un valor de N mayor de 100.000.000, y dado que su valor de R es el más cercano al Rmax/2 teórico, se puede concluir que, con los datos que tenemos:

- **Rmax/2** = 94.188.565,5
- **N1/2** = 100.000.000