INFORME
PRÁCTICA 1 & 2
ARQUITECTURA DE COMPUTADORES

Ibai Zak Allan Aldanondo 1397750
Ramon Guimerà Ortuño 1400214
Grupo: 419
Fecha: 10/03/2016
Departamento: CAOS

Indice

Indice de tablas y códigos

Objetivo de la práctica:

Principalmente consiste en familiarizarse con la compilación de un programa con diferentes compiladores y ver sus tiempos de respuesta sobre el mismo bucle pero con diferentes iteraciones.

Además de ver el efecto de código mal o no optimizado sobre iteraciones o un uso extenso de memoria en el tiempo de ejecución final.

Ejercicios:

Antes de la práctica se nos pidió hacer un previo a élla, evaluamos un mismo código sobre un compilador base y después era evaluado sobre el mismo compilador con diferentes opciones y posteriormente sobre un compilador distinto.
Sobre esos valores se hicieron 4 ejercicios de comparación de resultados.

5G = 50000 * 100000

1A)

	Op. Count	Op per Inst	IPC	Ciclos por Op	Speed Up
gcc v4.4	5G	0,0378	1,372	19,69	1
gcc -o3 v4.4	5G	0,0908	1,568	7,09	2,92
gcc -ofast v4.9	5G	0,998	1,338	0,75	27,29
icc -o3 v16.0	5G	2,2831	0,584	0,75	27,29

*Tabla ejercicio 1A

1B)
-o3 -> menos ciclos por operación y 3 veces más operaciones por instrucción.
-ofast -> casi una operación por instrucción.
icc -> entre 2 y 3 operaciones por instrucción.
Tanto -ofast y icc, tienen una disminución de ciclos por operación.

Por lo tanto, si debemos escoger entre gcc -ofast y icc -o3 para ver cuál es mejor, tanto uno como el otro son válidos debido a que su speedup son iguales y mucho superior a gcc -o3 y gcc. Pero si de verdad hemos de escoger entre uno y otro, deberíamos tener preferencia entre operaciones por instrucción o instrucciones por ciclo.

1C)

	Secuencial (seg)	Paralelo (seg)
gcc -o3 v4.4	21,02	4,5
gcc -ofast v4.9	15,014	0,47
icc -o3 v16.0	46,546	0,38

*Tabla ejercicio 1C

1D)
El mismo programa bajo distintos compiladores tenemos que el primero (icc), usa una funcion de comparacion por tamaño llamando a memoria cada vez, lo que es más costoso, pero menos instrucciones en total, mientras que el otro (gcc) usa funciones logicas de tipo int que son menos costosas que ir a memoria con el inconveniente de realizar más instrucciones.
El primero va a memoria 4 veces por iteración mientras que el otro 1.

icc Assembler	icc pseudo C
..B1.2	do{
maxps Vect(,%rdx,4), %xmm1	tmp1 = max(Vect[i], SZ);
maxps 16+Vect(,%rdx,4), %xmm1	tmp1 = max(Vect[i+1], SZ);
maxps 32+Vect(,%rdx,4), %xmm1	tmp1 = max(Vect[i+2], SZ);
maxps 48++Vect(,%rdx,4), %xmm1	tmp1 = max(Vect[i+3], SZ);
addq $16, %rdx	i += 4;
cmpq $50000, %rdx	c = (i<3125);
jb ..B1.2	}while(c);

*Tabla ejercicio 1D.A

gcc -o3 Assembler	-o3 Pseudo C
L.24	do{
movss Vect(%rax), %xmm1	tmp1 = Vect[i]
addq $4, %rax	i++;
movaps %xmm1, %xmm2	tmp2 = tmp1;
cmpq $200000, %rax	c = (i<50000);
movaps %xmm0, %xmm3	tmp3 = tmp1;
cmpless %xmm0, %xmm2	tmp2 = tmp2 - tmp0;
movaps %xmm2, %xmm0	tmp0 = tmp2;
andps %xmm2, %xmm3	tmp3 = AND(tmp3, tmp2);
andnps %xmm1, %xmm0	tmp0 = NAND(tmp0, tmp1);
orps %xmm3, %xmm0	tmp0 = OR(tmp0, tmp3);
jne .L24	}while(c);

*Tabla ejercicio 1D.B

1E)

	Op. Count	Op per Inst	IPC	Ciclos por Op
gcc -ofast	5G	0,0391	2,9	8,8
icc -o3	5G	1,7736	1,0303	0,5472

*Tabla ejercicio 1E.A

	CPU Time (seg.)	Porcentaje Tiempo Bucle
gcc -ofast	13,259	51%
icc -o3	0,783	31,70%

*Tabla ejercicio 1E.B

Podemos ver que icc proporciona una mejora consistente en lo que se refiere al tiempo total dentro de CPU además del tiempo dedicado dentro del bucle.

1F)

	Tiempo (seg.)
gcc normal	81,726
gcc nueva	0,1663

*Tabla ejercicio 1F

Con la nueva instrucción añadida, observamos claramente como el tiempo de ejecución es notablemente inferior a su ejecución “estándar”.

La instrucción que cuenta bits es:
sarq $0x1, -0x18(%brp)

En el ejercicio 2, teníamos como objetivo evaluar el mismo código sobre diferentes compiladores y sobre distintos tamaños de bucle. Obtenemos entonces la siguiente tabla:

2A)
MM = millones

	N	Checksum	Estimated Time (seg.)	Inst.	Ciclos
gcc -ofast	20K	3342994	4,411	28455 MM	14872 MM
	40K	133094104	17,93	114317 MM	60353 MM
	45K	168529120	362,96	162224 MM	112633 MM
icc -o3	20K	33429992	0,5	1340 MM	1714 MM
	40K	133093816	2,6	5958 MM	8832 MM
	45K	168529424	278,23	21532 MM	36735 MM

*Tabla ejercicio 2A

Tanto en icc -o3 como en gcc -ofast con 45K = N, el tiempo era mucho superior comparado a su ejecución anterior (N=40K) debido a los fallos de página y los content-switches ya que la variable no cabía dentro de la página.

2B)

	N	Checksum	Elapsed Time (seg.)	Inst.	Ciclos
icc	50K	207755792	0,57	4696 MM	1925 MM
	200K	3331527680	9,67	75054 MM	32623 MM
gcc	50K	207755808	15,82	115065 MM	53322 MM
	200K	3331511808	253,11	1840 MM	853404 MM

*Tabla ejercicio 2B

2C)
En el ejercicio 2C, primeramente se optimiza MapReduce.c sacando la variable tmp y juntando el doble bucle for en el segundo doble bucle for, es decir, donde aparece tmp en el segundo doble bucle, se sustituye por el valor de tmp calculado en el doble bucle anterior.

N	200k	200k	50k	200k	50k
File	MapReduce sin tmp	MapReduce.c sin tmp optimizado	MapReduce.c sin tmp optimizado	MapReduce.c sin tmp optimizado	MapReduce.c sin tmp optimizado
Compilador	icc -o2	icc -o3	icc -o2	icc -o2	gcc -02
Checksum	832889088	832889088	51935164	832880704	207755808
context-switches	16	11	3	35	13
page-faults	950	951	332	920	332
cycles	30629 MM	30705 MM	6367 MM	101894 MM	37784 MM
instructions	67545 MM	67545 MM	12209 MM	195157 MM	52551 MM
elapsed time	9,07	9,1	1,89	30,24	11,2

Tabla ejercicio 2C

Observamos como en la tabla tenemos MapReduce.c en distintos compiladores con distintas optimizaciones. En la última columna, se esperaba un Elapsed Time de un máximo aproximado de 3 segundos, por causas desconocidas incluso con la ayuda del profesor ante el código, no pudimos conseguir bajar de los 11,2 segundos.

Se adjunt además las optimizaciones de código realizadas:

Columna 2 y 3 (código 2C.1):

```
float Rj0;
float Rj1;
float Rj2;
float Rj3;
float V1j0;
float V2j0;
float V1j1;
float V2j1;
float V1j2;
float V2j2;
float V1j3;
float V2j3;

for(j=0; j<N; j+=4)
{
	Rj0 = 0.0f;
	Rj1 = 0.0f;
	Rj2 = 0.0f;
	Rj3 = 0.0f;
	V1j0 = V1[j];
	V2j0 = V2[j];
	V1j1 = V1[j+1];
	V2j1 = V2[j+1];
	V1j2 = V1[j+2];
	V2j2 = V2[j+2];
	V1j3 = V1[j+3];
	V2j3 = V2[j+3];

	for(i=o; i<N; i+=4)
	{
		Rj0 = Rj0 + (X[i]-V1j0)*(X[i]-V2j0);
		Rj1 = Rj1 + (X[i+1]-V1j1)*(X[i+1]-V2j1);
		Rj2 = Rj2 + (X[i+2]-V1j2)*(X[i+2]-V2j2);
		Rj3 = Rj3 + (X[i+3]-V1j3)*(X[i+3]-V2j3);
	}
	R[j] = Rj0;
	R[j+1] = Rj1;
	R[j+2] = Rj2;
	R[j+3] = Rj3;
}
```

Viendo que este código no proporcionaba ninguna mejora, al contrario, entregaba resultados erróneos, se porcedió a modificarlo, quedando entonces el siguiente código:

Columna 5 (código 2C.2):

```
float Rj;
float V1j;
float V2j;

for(j=0; j<N; j+=1)
{
	Rj = 0.0f;
	V1j = V1[j];
	V2j = V2[j];

	for(i=0; i<N; i+=1)
	{
	Rj = Rj + (X[i]-V1j)*(X[i]-V2j);
	}
	R[j] = Rj;
}
```

El código anterior, supuestamente debería darnos una mejora en el elapsed time, bajándalo por abajo de los 3 segundos de ejecución.

Conclusiones:

Después de completar estas sesiones de prácticas, podemos concluir y asegurar que un mismo código bajo diferentes compiladores difiere, en gran parte, en sus tiempos de ejecución, fallos de página, etc. proporcionando sin embargo, los mismo resultados esperados del programa.

Además, la optimización correcta de un programa nos ha permitido observar como bajo el mismo compilador y la misma máquina nos entregaba resultados de tiempo de ejecucion, page-faults, etc. mejores que con el código no optimizado.

Cabe decir que en el ultimo ejercicio no conseguimos el objetivo por razones desconocidas a nuestro conociemiento y también desconocidas por el profesor de prácticas encargado de la sesión.