Laboratorio Sesión 01: Compilación y rendimiento

Objetivo

El objetivo de esta sesión es introducir el uso de diferentes parámetros de compilación y su influencia en el rendimiento de un programa, así como formas sencillas de evaluar dicho rendimiento.

Conocimientos Previos

Parámetros de compilación de un programa

Habitualmente cuando compilamos un programa mediante un compilador genérico (en esta asignatura utilizaremos *gcc*) acostumbramos a dejar en sus manos todas las decisiones. Básicamente decimos: "Este es mi programa en alto nivel, dame el programa compilado". Aunque esta forma de trabajar es sin duda bastante cómoda y adecuada en la mayoría de los casos, también implica renunciar a decirle al compilador qué tiene que hacer y qué no para obtener el resultado que nosotros deseamos. En esta práctica veremos algunos parámetros de compilación que pueden hacer que el programa compilado resultante se ajuste más a nuestras necesidades en algunas situaciones concretas que podemos encontrar como ingenieros.

Para compilar un programa en primer lugar deberemos partir de los ficheros fuente (supongamos que se llaman *miprograma_parte1.c* y *miprograma_parte2.c*) y deberemos obtener el código ensamblador equivalente:

```
gcc -S miprograma_parte1.c miprograma_parte2.c
```

Con esto obtendremos los ficheros *miprograma_parte1.s* y *miprograma_parte2.s* que contendrán el código ensamblador de los anteriores programas. Para obtener con ellos el código objeto usaremos cualquiera de los siguientes comandos equivalentes:

```
as miprograma_parte1.s miprograma_parte2.s
gcc -c miprograma_parte1.s miprograma_parte2.s
```

Y obtendremos los ficheros *miprograma_parte1.o* y *miprograma_parte2.o* que contendrán el código objeto equivalente. Finalmente deberemos juntar (*linkar*) el código de ambos programas y generar uno solo (como en el caso anterior, los siguientes dos comandos son equivalentes):

```
ld -o miprograma miprograma_parte1.o miprograma_parte2.o
gcc -o miprograma miprograma_parte1.o miprograma_parte2.o
```

Con lo que obtendremos el ejecutable *miprograma*. También podemos realizar los tres pasos directamente:

```
gcc -o miprograma miprograma_parte1.c miprograma_parte2.c
```

Por cualquiera de los dos caminos el compilador genera el programa *miprograma* en el mínimo tiempo de compilación dando lugar a un ejecutable que es correcto y cómodo para debuggar el programa, ya que es una traducción bastante literal del código fuente original. Sin embargo, muchas veces al compilar un programa no querremos que la salida sea cómoda para debuggar, sino que buscaremos que el código compilado sea rápido. Para ello debemos activar las optimizaciones con la línea:

```
gcc -O -o miprograma miprograma.c
```

El compilador tratará de optimizar el código generado. Existen 4 niveles de optimización: `-O0 -O1 -O2 -O3`. El primero equivale a no optimizar y el segundo aplica las optimizaciones básicas (equivale a `-O`). Con `-O2` el compilador aplicará optimizaciones avanzadas que pueden llevarle bastante tiempo (aunque no en programas sencillos como los de esta práctica). Finalmente con `-O3` el compilador aplicará optimizaciones arriesgadas que pueden llegar a relentizar el programa. Tenéis toda la información en el manual de gcc.

Consideraciones prácticas a la hora de medir el rendimiento

A la hora de medir el rendimiento real de un programa, habitualmente nos encontramos con una serie de problemas que no son tan simples como puede parecer. Veamos la idea: ¿cómo se sabe cómo es de rápido un programa? Pues midiendo cuanto tarda en ejecutarse. Claro, el primer problema práctico es ¿cómo se mide cuánto tarda un programa? ¿podemos utilizar un cronómetro como haríamos con un coche? Pensemos un poco en las magnitudes de las que hablamos. Un procesador actual puede funcionar sin problemas a una velocidad de 1GHz (de hecho a más, pero con 1 GHz es más fácil hacer los cálculos) así que esto significa que tendrá un reloj interno que generará 1000000000 de ciclos cada segundo. ¿Cuántas instrucciones puede ejecutar cada ciclo? Esta respuesta no es fácil, pero podemos decir que, más o menos, entre media y cuatro. Depende mucho del programa y del procesador. En cualquier caso, si suponemos una instrucción por ciclo (más o menos razonable) y 1GHz estaremos hablado que cada **milmillonésima** de segundo el procesador ejecuta una instrucción. Si tenemos un programa corto (1000 instrucciones) estaríamos hablando de que tardaría del orden de una millonésima de segundo en ejecutarse. Está claro que un cronómetro no es suficiente. En general, el mejor método para medir tiempos es usar el propio reloj de sistema del procesador, pero ya veréis que tampoco es muy preciso, así que lo mejor es medir programas largos o ejecutarlos muchas veces y hacer una media.

Otra posibilidad para medir cuánto tarda un programa es utilizar una medida más propia de los procesadores que los segundos: los ticks de reloj. Aunque estas dos medidas parecen equivalentes, en realidad esto no es cierto ya que la relación "el procesador va a 1 GHZ y por tanto cada milmillonésima de segundo hace un ciclo de reloj" dejó de ser cierta hace años. Los procesadores ajustan dinámicamente su velocidad según la temperatura y la carga de trabajo. Esto significa que la única forma realista de saber cuantos ticks de reloj pasan desde que se empieza a ejecutar un programa hasta que acaba es preguntarle al propio procesador. Esta medida es fiable en el sentido en que la respuesta es un número exacto. Sin embargo, un mismo programa determinista ejecutado varias veces no siempre tarda lo mismo. ¿por qué? Pues debido a los fallos de caché/memoria, las interrupciones, el tiempo que tarda el sistema operativo, etc.

Finalmente la tercera medida interesante para evaluar un programa es la cantidad de instrucciones que ejecuta (instrucciones dinámicas). Para contar cuantas instrucciones ejecuta un procesador en un programa tenemos varios métodos:

- **Cálculo directo.** Es la forma más rápida para códigos simples y la más complicada para programas largos. Consiste, básicamente, en hacer la cuenta teórica: "si mi programa tiene un bucle que da 1000 vueltas y cada vuelta ejecuta 5 instrucciones y fuera del bucle hay otras 50 instrucciones, en total, mi programa ejecuta 5050 instrucciones". Para programas complejos se complica porque los condicionales no suelen tener las mismas instrucciones en cada rama, los bucles no siempre son determinístas, las subrutinas son difíciles de contar y los programas ejecutan diferentes instrucciones para diferentes entradas. Para programas medios es inaplicable.

- **Contadores Hardware.** Este sistema consiste en acceder directamente a diferentes registros contador internos al procesador que nos pueden decir, entre otras cosas, cuántas instrucciones se ejecutan, cuántos saltos se han predicho correctamente o cuántos fallos de caché ha habido. Es a la vez un sistema rápido y fiable ya que es el propio procesador el que informa de su estado y, además, al ser contadores hardware, el programa se ejecuta a su velocidad original. Como inconvenientes están que no todos los procesadores tienen contadores hardware (o no tienen los mismos o no se accede de la misma forma), que es necesario recompilar el kernel de Linux para poder acceder a ellos y que este sistema no permite alterar el código original para verificar cambios (a no ser que tengamos los fuentes y recompilemos).

- **Instrumentar el código.** Instrumentar el código es un sistema que consiste en añadir al código ejecutable del programa (o bien en la compilación o bien en la ejecución) nuevas instrucciones (directamente en código máquina) que se encargan de realizar las medidas deseadas, por ejemplo, contar las instrucciones ejecutadas o averiguar los datos que se envían a la memoria. Los mayores inconvenientes de este método son que

al añadir más instrucciones el código se ejecuta más lento y que, además, el código que se ejecuta no es el original sino el modificado. Sin embargo este sistema permite realizar numerosos tipos de medidas e, incluso, modificar el funcionamiento del programa cambiandole unas instrucciones por otras.

- **Simular el procesador.** Este último método es el más complejo y, a la vez, el que permite mayor control del sistema. Consiste en crear un programa software que "emula" o "simula" (la diferencia podéis buscarla en Internet) el comportamiento del procesador que queremos estudiar. Este programa es el que lee el código compilado y "ejecuta" el programa que queremos estudiar. Este sistema es con mucha diferencia el más lento y precisa tener un conocimiento muy detallado de cómo funciona el procesador a estudiar pero, por otro lado, permite saber el resultado de cualquier cambio en el procesador (como por ejemplo averiguar si un nuevo multiplicador que es más lento pero realiza más multiplicaciones en paralelo hace que los programas se ejecuten más rápido o no).

En esta práctica para medir el rendimiento de los programas utilizaremos una mezcla entre pedirle información al sistema operativo (y al procesador) e instrumentar el código. Para obtener el número de instrucciones dinámicas utilizaremos el programa de instrumentación *Valgrind*. Para ejecutarlo deberéis escribir:

```
valgrind --tool=lackey ./miejecutable
```

Entre toda la información que imprime nos interesa la que se encuentra bajo el epígrafe `guest instrs` que corresponde a las instrucciones ejecutadas en el código ensamblador del procesador.

Para medir el tiempo de ejecución o los ciclos utilizaremos dos códigos: el programa `tiempo.c` y la librería `cycle.h`. El primero contiene la rutina `GetTime()` que devuelve un *float* con el tiempo actual en milisegundos. La segunda contiene la rutina `getticks()` que devuelve una variable de tipo *ticks* que se puede usar como entrada para la rutina `elapsed()` que devuelve un *double* con los tics de reloj que han pasado en la ejecución de un programa. Recordad que en el primer caso se debe incluir el nombre del fichero fuente en la orden de compilación (por ejemplo, `gcc -o simple simple.c tiempo.c`). A continuación se muestran sendos ejemplos de cómo medir el tiempo y los tics en un determinado código:

Medir tiempo

```
long long i,resultado=0;
float t1,t2;
t1=GetTime();

// Inicio del programa a medir
for (i=0; i<1000000;i++)
   resultado= resultado+i;
// Fin del programa a medir

t2=GetTime();
printf("Milisegundos = %9f\n",
                       t2-t1);
```

Medir ciclos

```
long long i,resultado=0;
ticks t1,t2;
t1=getticks();

// Inicio del programa a medir
for (i=0; i<1000000;i++)
   resultado= resultado+i;
// Fin del programa a medir

t2=getticks();
printf("Ciclos = %lf\n",
                  elapsed(t2,t1));
```

Aunque evidentemente un programa con el código para medir tiempos ejecuta más instrucciones dinámicas que uno que no lo contiene y, a su vez, un código que está contando las instrucciones dinámicas tarda más en ejecutarse que uno que no, la influencia en estas prácticas es pequeña, así que ambas medidas pueden hacerse a la vez. Eso sí, no en la misma ejecución donde se usa `valgrind`.

Finalmente el último parámetro a tener en cuenta cuando se están midiendo tiempos o ciclos es que en los procesadores actuales de múltiples núcleos existe la posibilidad de que un programa migre de un procesador a otro. Para evitar que este efecto influya en nuestras medidas nos aseguraremos de que el programa se ejecuta con afinidad a un solo procesador con la orden:

```
taskset -c 1 miprograma
```

Si además tenemos acceso de *root* al sistema (es decir, esto NO aplica al laboratorio, es puramente informativo) puede ser bueno asignar la máxima prioridad a nuestro programa de forma que el Sistema Operativo no lo interrumpa con otro programa:

```
chrt -f 99 miprograma
```

Lo que si se puede hacer es darle al programa diferentes prioridades de usuario, siendo la -19 la más favorable:

```
nice -19 miprograma
```

MIPS

Una vez que somos capaces de medir las instrucciones y el tiempo que tarda en ejecutarse un programa podemos obtener con facilidad los MIPS (Millones de Instrucciones Por Segundo) que realiza el procesador. La fórmula es simple:

$$MIPS = \frac{\#instrs}{10^6 * segundos}$$

Sin embargo hay que tener en cuenta que esta medida depende mucho del procesador (si varía el ISA varían los MIPS), del compilador (dos compiladores distintos no generan el mismo código y por tanto no se puede comparar) y de las optimizaciones (que un programa esté más optimizado no significa que el procesador sea más rápido). Así que en realidad no es muy fiable a la hora de comparar procesadores distintos. Si que sirven en cambio para comparar procesadores de la misma familia e incluso para comparar sistemas. Linux, por ejemplo, siempre calcula los MIPS en tiempo de arranque del sistema. Podéis comprobarlo ejecutando:

```
dmesg | grep MIPS
```

Si tenéis curiosidad sobre la importancia de los MIPS y el uso de los BogoMIPS podéis consultar:

```
http://es.tldp.org/COMO-INSFLUG/COMOs/BogoMIPS-mini-Como/BogoMIPS-
       mini-Como-8.html
```

Cómo hacer las medias

Finalmente, un punto importante a la hora de realizar todas estas medidas es decidir cómo obtener las medias. Como ya se ha comentado, el tiempo e incluso los ciclos de ejecución de un programa están sujetos a una fuerte variabilidad. Para reducir este efecto, cuando toméis medidas variables deberíais hacer una media de las tres medidas más "razonables" de una muestra de cinco. Es decir: realizaremos la medida deseada 5 veces y a continuación descartaremos el valor más alto y el más bajo. Finalmente, el valor buscado será la media de las tres medidas restantes. Por motivos de tiempo, no es necesario que lo hagáis así en esta práctica pero tenedlo en cuenta en adelante cuando os enfrentéis a resultados muy variables.

Estudio Previo

1. Busca que significa hacer "inlining" de una función.
2. Busca que opción de compilación individual (no grupos de opciones como -O) de *gcc* permite al compilador hacer "inlining" de todas las funciones simples. Averigua si esta opción se activa al activar las optimizaciones -O2 del compilador. ¿Para qué sirve la opción -finline-limit?

3. Explica una forma práctica de saber si en un programa ensamblador existe la función "Juanito". Explica cómo averiguar si, además de existir, esa función es invocada o no.

4. Dado el siguiente segmento de código y su traducción a ensamblador (compilado con gcc):

Código C	Código ASM
	`8048515: mov 0x5c(%esp),%eax`
`int i,resultado=0;`	`add %eax,0x58(%esp)`
`for (i=0; i<1000000;i++)`	`addl $0x1,0x5c(%esp)`
`resultado= resultado+i;`	`cmpl $0xf423f,0x5c(%esp)`
	`jle 8048515 <main+0x37>`

   Calcula cuántas instrucciones estáticas y dinámicas tiene el anterior segmento de código. Si la ejecución tarda 14 ms y 16000000 de ciclos, calcula cuántos MIPS y que IPC, CPI y frecuencia tiene el procesador al ejecutar este código.

5. A continuación analiza el mismo segmento de código compilado con la opción -O:

Código ASM (-O)

```
80484b8: add    %eax,%ebx
add    $0x1,%eax
cmp    $0xf4240,%eax
jne    80484b8 <main+0x28>
```

   En este caso el programa tarda 7 ms y 8000000 de ciclos en ejecutarse. Calcula nuevamente los MIPS, el CPI y la frecuencia del procesador e indica cuál es el Speedup respecto a la versión anterior. Intenta explicar cuáles son las posibles fuentes de las igualdades y diferencias observadas con respecto al apartado anterior.

6. Si el código anterior es parte de un programa que tarda en ejecutarse (en total) 200 ms (compilado todo él con la opción -O0), calcula cuál es el Speedup máximo del programa total si consiguiéramos ejecutar el código anterior de manera instantánea. Calcula también el Speedup del programa completo obtenido al compilar solo el código del ejemplo con la opción -O.

7. A partir de las herramientas de medida prácticas que se han visto en los apartados anteriores define una forma (qué medir, cómo y qué hacer con el número) para medir el rendimiento (MIPS y CPI) del programa en C que acabamos de ver. Aunque un programa donde se miden los ciclos y el tiempo de ejecución ejecuta más instrucciones que uno que no, no es necesario que tengáis esto en cuenta en esta práctica ya que el efecto es muy pequeño.

8. Un programa dado lo hemos ejecutado 5 veces con los siguientes resultados de tiempo de ejecución: 10 ms, 8ms, 13 ms, 15ms y 2ms. Calcula la media aritmética y geométrica de las 5 medidas. A continuación descarta las dos medidas extremas y calcula de nuevo la media geométrica y aritmética de los resultados. Explica cuales son los principales efectos que observáis.

Trabajo a realizar durante la Práctica

1. Dado el programa en C con el código utilizado durante los ejemplos de este documento y que al final imprime el valor de la variable "resultado", `Simple.c`, compílalo y ejecútalo midiendo las instrucciones dinámicas. A continuación compílalo de forma optimizada (-O2) y vuelve a medir las instrucciones dinámicas que ejecuta. Explica los resultados.

2. A continuación trabajaremos el código del programa `Poker.c`. Utiliza las herramientas descritas para medir los MIPS, el CPI y la frecuencia de trabajo del procesador cuando se ejecuta dicho programa compilado sin optimizaciones.

   A continuación recompila el programa optimizando (-O2) y vuelve a calcular los parámetros anteriores. Calcula cual es el Speedup del programa.

   Compila de nuevo el programa obligándolo a que, además de optimizar con -O2, haga "inlining" de todas las rutinas simples (ajustando el límite de "inlining" en 500) y calcula sus nuevos parámetros de rendimiento.

3. Centraos ahora sólo en el código que se ejecuta en la rutina "PierdeTiempo". Calcula qué porcentaje de los ciclos de ejecución del programa total (compilado con -O0) se pierde en ejecutar esta rutina (tened cuidado de no contar también el tiempo del `printf` en los ciclos de la rutina; para evitarlo podéis sacar la rutina de la llamada al `printf` asignando su resultado a una variable temporal que después se imprima). Calcula el Speedup máximo teórico del programa que se podría alcanzar eliminando esta rutina del código. Repite los cálculos compilando con la opción -O2. Explica a que se deben las variaciones.

Nombre:_____ Grupo:_____

Nombre:_____

Hoja de respuesta al Estudio Previo

1. Hacer "inlining" de una función significa:

2. La opción específica de compilación de *gcc* que permite al compilador hacer "inlining" de todas las funciones simples es (especifica si se activa o no al activar la opción -O2). ¿Para qué sirve la opción -finline-limit?:

3. Explica una forma práctica de saber si en un programa ensamblador existe la función "Juanito" y cómo averiguar si, además de existir, esa función es invocada o no:

4. El primer código ensamblador tiene:

   Instr. estáticas: ____ Instr. dinámicas: ____

   Si la ejecución tarda 14 ms y 16000000 de ciclos:

   MIPS: ____ IPC: ____ CPI: ____

   Frecuencia: ____

5. El segundo código (compilado con -O) tiene:

   Instr. estáticas: ____ Instr. dinámicas: ____

   Si la ejecución tarda 7 ms y 8000000 de ciclos:

   MIPS: ____ CPI: ____ Frecuencia: ____

   Speedup: ____

   Las igualdades y diferencias observadas respecto al apartado anterior se deben a:

6. El programa total puede obtener un Speedup de:

   Si el código es instantáneo: [ ] Si se compila con -O: [ ]

7. Una forma práctica para medir el rendimiento (MIPS e IPC) del programa en C que acabamos de ver es:

   [ ]

8. Dadas 5 ejecuciones de 10 ms, 8ms, 13 ms, 15ms y 2ms. Su media:

   Geométrica: [ ] Aritmética: [ ]

   Descartando los valores extremos su media es:

   Geométrica: [ ] Aritmética: [ ]

   Se observa que:

   [ ]

Nombre:_____ Grupo:_____

Nombre:_____

Hoja de respuestas de la práctica

1. Instrucciones dinámicas del código de ejemplo `Simple.c`:

   Sin optimizar: [ ] Optimizado: [ ]

   Explicación de los resultados:

   _____

   _____

   _____

2. Rellena la siguiente tabla sobre el programa `Poker.c`:

	Tiempo	Ciclos	Instrucciones	MIPS	CPI	Frec.	Speedup
-O0							
-O2							
-O2 + "inlining"							

3. Rellena la siguiente tabla sobre la rutina `PierdeTiempo` (tened cuidado de no incluir el `printf` en los ciclos de la rutina) del programa `Poker.c`:

	Ciclos Programa	Ciclos rutina	% de Ciclos	Speedup teórico máximo
-O0				
-O2				

   Las variaciones entre las dos filas del apartado anterior se deben a:

   _____

   _____

   _____