Guión de la sesión 10

Introducción

La tecnología de desarrollo de los compiladores hace hoy posible obtener un alto grado de optimización en el código generado, haciendo completamente innecesario trabajar a nivel ensamblador de forma manual.

Sin embargo, las buenas soluciones siguen lejos de obtenerse con la simple pulsación de un botón. Por ejemplo, muchas de las optimizaciones que puede realizar el compilador deben indicarse expresamente entre los parámetros de la compilación, y no se incluyen como opciones por defecto. En otras ocasiones el patrón de acceso a la memoria utilizado por la aplicación no se adapta a la arquitectura sobre la que debe ejecutarse el código, por lo que la mejora está fuera del alcance del propio compilador.

El propósito de esta práctica es entender el funcionamiento del compilador y la manera en la que las funciones se ejecutan sobre la arquitectura. Sigue para ello el guión que se describe a continuación, y contesta a cada una de las cuestiones que se remarcan en negrita.

Objetivos

• Entender la relación entre el código binario del programa y la arquitectura a través del ABI (Application Binary Interface)

• Analizar el efecto de las optimizaciones del compilador sobre el código ensamblador generado

Procedimiento

Caso de estudio: Suma de los elementos de un vector de N posiciones

El código de partida simplemente recorre una a una las posiciones de un vector, acumulando el resultado sobre una variable total. Este cálculo se repite 100 veces seguidas.

1. Abre el fichero array.c y observa la estructura del código, particularmente la función add_array. Observa cómo la variable local result acumula el resultado de la suma de los elementos del vector.
2. Compila ahora el programa y genera 2 ejecutables distintos:
   - ➔ El primero sin incluir opciones de optimización. En el caso de gcc ejecuta: gcc -g array.c -o array_O0 -O0

     Donde –g se utiliza para incluir información de depuración, utilizada posteriormente para asociar el ensamblador generado con el código C original, y-O0 desactiva cualquier tipo de optimización.
   - ➔ El segundo indicando optimización mínima (-O1).
3. Genera ahora el desensamblado correspondiente a las versiones sin optimización y con optimización mínima, para analizar las diferencias entre uno y otro:

objdump -S ./array_O0 > array_O0.S

objdump -S ./array_O1 > array_O1.S

4. Analiza ahora las diferencias entre ambos códigos, pero solamente en lo que se refiere a la función add_array, en los siguientes 2 aspectos: el uso de la pila, y el número de accesos a la memoria (ten en cuenta que la pila se ubica en memoria).

➔ El uso de la pila siempre se inicia con una instrucción del tipo

push %rbp

- ¿Hacen uso de la pila ambas versiones?
- Averigua el tamaño de datos de la variable local y los argumentos. ¿Cómo se pasan a la función los argumentos? ¿Dónde se almacena la variable local? En el caso del uso de la pila dibuja la estructura utilizada por la función, ubicando cada variable o argumento en la dirección mostrada por el código.

➔ Busca ahora el cuerpo del bucle, e identifica las instrucciones que realizan acceso ala memoria dentro de él, con cualquiera de los siguientes patrones:

mov<sufijo> <registro>, <constante>(<registro>)

mov<sufijo> <constante>(<registro>), <registro>

push <registro>

pop <registro>

La instrucción lea “Load Effective Adress” sirve para calcular la dirección de acceso a la memoria, y en algunos casos precede el mov de almacenamiento, pero no siempre. El significado del formato el es siguiente:

lea <offset>(<reg1>,<reg2>,<constante>),

<reg3> reg3 <= reg2 * constante + reg1 + offset

➔ ¿Qué registros se utilizan en la suma, y qué operandos contienen? ¿Cuál es el tamaño de datos de estos registros?.

➔ ¿Cuántos accesos a memoria se realizan en cada caso? Distingue los de lectura de los de escritura.