

Índice
Objetivos de la sesión
Conocimientos y materiales necesarios
1. Obtención de los parámetros de caché
2. Influencia de la localidad en el rendimiento
3. Simulación de los accesos a memoria con Valgrind
Archivos de la práctica
Ejercicios

Análisis de cachés reales

Área de Arquitectura y Tecnología de Computadores – Versión 1.2.12026, 23/10/2023

Objetivos de la sesión

El objetivo de esta sesión práctica es mostrar la influencia de la caché y del estilo de programación sobre el rendimiento en computadores reales.

Conocimientos y materiales necesarios

Para obtener el máximo aprovechamiento de esta sesión es necesario que el alumno:

- Disponga de una máquina, física o virtual, con un sistema operativo Ubuntu Server 16.04 LTS sobre el que se trabajará a lo largo del curso.
- Durante la sesión de plantearán una serie de preguntas que puedes responder en el correspondiente [cuestionario](#) en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `<Ctrl>`.

1. Obtención de los parámetros de caché

En este apartado se obtendrán los parámetros de la caché del computador sobre el cual se realiza la sesión práctica. Estos parámetros son importantes para explicar el comportamiento temporal de los diferentes programas que se emplearán a lo largo de la sesión. Además, permitirán conocer los parámetros de caché de computadores reales, pues muchas veces en las clases teóricas se trabaja con cachés muy simplificadas para poder ser representadas en una hoja de papel.



- Descarga los ficheros de la práctica y descomprímelos con los siguientes comandos:

```
$> wget https://www.atc.uniovi.es/grado/2ac/files/session3-3.tar.gz
$> tar xvfz session3-3.tar.gz
```
- Entra dentro del directorio `bandwidth-1.3.1` que contiene un benchmark de memoria y compila la versión de 32 bits:


```
$> make bandwidth32
```
- Aparecen errores ya que se requiere el `nasm`, un ensamblador para Linux.
- Instala `nasm`:

```
$> sudo apt install nasm
```
- Intenta de nuevo la compilación.
- Ejecuta el programa `bandwidth` con el siguiente comando:

```
$> ./bandwidth32 | more
```
- Indica los diferentes niveles de caché. Para cada nivel incluye: tamaño, estrategia de correspondencia, tamaño del bloque, si es unificada o por el contrario si es de código o datos, así como el número de cachés de dicho tipo. Además, debes tener en cuenta que en los procesadores multinúcleo es habitual replicar los niveles de caché más bajos en todos los núcleos. Responde en el [cuestionario](#): pregunta 1.
- Interrompe la ejecución del programa con la combinación de teclas `<Ctrl>`+`<C>`.
- Ejecuta de nuevo el benchmark con el siguiente comando:

```
$> sudo nice -n -2 ./bandwidth32 --fastest
```
- El programa `nice` se utiliza para ejecutar el benchmark a mayor prioridad para que no interfieran los servicios del sistema.

El benchmark utiliza distintas formas de escribir y leer la memoria. Además varía el tamaño de los datos con los que trabaja. Esto hace que a medida que el tamaño de esos datos aumenta aparezcan fallos de caché. Cuando los datos son muy grandes ya no entran en L1 y se tiene que usar L2. Esto tiene una penalización en el tiempo y se reduce el ancho de banda. Lo mismo ocurre cuando se sobrepasa la L2 y la L3.



- Espera a que se termine de ejecutar el programa. Debería tardar unos 5 minutos.
- Tras la ejecución se genera una gráfica en el fichero `bandwidth.bmp`. Transfiere ese fichero a tu máquina local y analiza el resultado. En el eje Y se muestra el ancho de banda y en el eje X se muestra el tamaño de los datos con los que se trabaja.
- No te fijas en los experimentos concretos sino en las tendencias. Deberían aparecer escalones que encajen aproximadamente con el tamaño de las cachés del sistema.
- Indica el ancho de banda para el experimento de lectura secuencial de 128 bits para todos los niveles de caché y la memoria principal. Responde en el [cuestionario](#): pregunta 2.
- ¿Cuántas veces es más rápida la memoria caché L1 que la memoria principal? Responde en el [cuestionario](#): pregunta 3.

La memoria caché L2 es mucho más rápida que la memoria principal, pero algo más lenta que la caché L1, a pesar de estar hecha con la misma tecnología. La diferencia radica en que al ser mayor su tamaño, su tiempo de respuesta es también mayor. Por ejemplo, al tener más conjuntos, hace falta más tiempo para localizar el conjunto en el cual puede estar cacheada una palabra a partir de su dirección de memoria.



- ¿Cuántas veces es más rápida la memoria caché L2 que la memoria principal? Responde en el [cuestionario](#): pregunta 4.

2. Influencia de la localidad en el rendimiento

Vamos a analizar la influencia del estilo de programación sobre la localidad de los programas y por lo tanto sobre su tiempo de ejecución.



- Copia a tu carpeta de trabajo el programa `3-3loc1.c`, cuyo código se muestra en el listado siguiente.

```
1 #include <stdint.h>
2
3 #define NROWS      8192 // 2^13 rows
4 #define NCOLS      8192 // 2^13 cols
5 #define NTIMES     10 // Repeat 10 times
6
7 // Matrix size 2^26 = 64 MiBytes
8 uint8_t matrix[NROWS][NCOLS];
9
10 int main(void)
11 {
12     int i, j, rep;
13
14     for (rep = 0; rep < NTIMES; rep++)
15     {
16         for (i = 0; i < NROWS; i++)
17         {
18             for(j = 0; j < NCOLS; j++)
19             {
20                 matrix[i][j] = 0;
21             }
22         }
23     }
24 }
```
- Compila y enlaza el programa anterior con la orden `gcc 3-3loc1.c -o 3-3loc1`.
- Ejecuta el programa anterior con la orden `time ./3-3loc1`.
- ¿Cuál es el tiempo de ejecución del programa `3-3loc1`? Responde en el [cuestionario](#): pregunta 5.
- Para obtener una medida más fiable del tiempo de ejecución, ejecuta otras dos veces la orden `time ./3-3loc1` y quédate con el tiempo de ejecución total (`Real`) medio.
- A continuación debes estudiar la secuencia de accesos a la matriz, para lo cual debes introducir en el código fuente del programa, justo a continuación de la instrucción `matrix[i][j] = 0;`, el siguiente código y guardar el archivo resultante como `3-3loc2.c`:

```
1 // Prints the address accessed in the first iteration
2 // for the first 4 rows
3 if (rep == 0 && i < 4)
4 {
5     if (j == 0)
6         printf("Begin row\n");
7
8     printf("%p\n", &(matrix[i][j]));
9 }
```
- El código anterior escribe las direcciones de acceso a la matriz y además marca con la cadena `Begin row` el comienzo de cada fila para la primera repetición y las cuatro primeras filas. Recuerda además incluir la directiva `#include <stdio.h>` para poder emplear la función `printf`.
- Compila y enlaza el programa anterior. Tras esto, ejecútalo empleando la orden `./3-3loc2 > result2.txt`.
- Edita el archivo `result2.txt`. Observa la secuencia de direcciones de acceso, prestando especial atención a la última dirección de cada fila y a la primera de la fila siguiente. Puedes localizar fácilmente el final de una fila buscando la cadena `Begin row` dentro del editor. ¿Qué relación hay entre ambas direcciones? Responde en el [cuestionario](#): pregunta 6.
- Extrapolando la observación anterior, ¿qué ocurre con las direcciones de acceso a todos los elementos de la matriz? Responde en el [cuestionario](#): pregunta 7.

Ahora vamos a estimar la localidad del programa `3-3loc1.c` en el acceso a los elementos de la matriz. Una forma de hacerlo es obtener la tasa de aciertos de caché para una caché que se toma como referencia. A mayor tasa de aciertos, mayor localidad. Puesto que solo estamos interesados en la localidad en el acceso a los datos, ignoremos el efecto de la ejecución del sistema operativo y de otras tareas que también se encuentran en memoria. También ignoremos el efecto del código de la tarea, que también se encuentra en memoria.

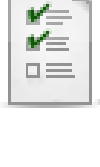


- Bajo los supuestos anteriores, ¿cuál sería la tasa de aciertos de caché L1? Debes suponer además que justo antes de acceder al primer elemento de la matriz la caché está vacía. Responde en el [cuestionario](#): pregunta 8.
- Para observar el efecto del estilo de programación sobre el tiempo de ejecución de los programas, debes modificar el programa `3-3loc1.c`, cambiando de orden las dos instrucciones que iteran sobre filas y columnas, de tal forma que primero se recorran todas las filas para una misma columna, antes de pasar a la siguiente columna, tal como se indica a continuación:

```
1 // Print address for first repetition
2 // and the first 4 rows
3 for (j = 0; j < NCOLS; j++)
4 {
5     for (i = 0; i < NROWS; i++)
6     {
7         matrix[i][j] = 0;
8     }
9 }
```
- Guarda el nuevo archivo con el nombre `3-3loc3.c`.
- Compila, enlaza y ejecuta el programa anterior empleando la orden `time ./3-3loc3`.
- ¿Cuál es el tiempo de ejecución del programa `3-3loc3`? Responde en el [cuestionario](#): pregunta 9.
- Para obtener una medida más fiable del tiempo de ejecución, ejecuta otras dos veces la orden `time ./3-3loc3` y quédate con el tiempo de ejecución total intermedio.
- Compara el tiempo medido con el del programa `3-3loc1`. ¿Qué ha ocurrido? ¿Qué explicación encuentras? Responde en el [cuestionario](#): pregunta 10.
- Ahora debes estudiar la secuencia de accesos a la matriz, tras haber cambiado el orden de acceso a la misma. Debes introducir en el código fuente del programa `3-3loc3.c`, justo a continuación de la instrucción `matrix[i][j] = 0;`, el siguiente código y guardar el archivo resultante como `3-3loc4.c`:


```
1 // Print address for first repetition
2 // and the first 4 columns
3 if (rep == 0 && j < 4)
4 {
5     if (i == 0)
6         printf("Begin column\n");
7
8     printf("%p\n", &(matrix[i][j]));
9 }
```
- El código anterior escribe para la primera repetición y las cuatro primeras columnas la dirección de comienzo de cada columna. Recuerda además incluir la directiva `#include <stdio.h>` para poder emplear la función `printf`.
- Compila, enlaza y ejecuta el programa anterior empleando ahora la orden `time ./3-3loc4 > result4.txt`.
- Edita el archivo `result4.txt` y observa la secuencia de direcciones de acceso, prestando especial atención a la última dirección de la primera columna y a la primera dirección de la segunda columna. ¿Qué ocurre con las direcciones de acceso a todos los elementos de la matriz? Responde en el [cuestionario](#): pregunta 11.

Vamos a estimar la localidad para este programa, tal como lo hiciste anteriormente. Debes ignorar el efecto de la ejecución del sistema operativo y de otras tareas que también se encuentran en memoria. También debes ignorar el código de la tarea, que también se encuentra en memoria. Debes suponer que justo antes de acceder al primer elemento de la matriz la caché está vacía.



- Bajo los supuestos anteriores, ¿cuál sería la tasa de aciertos de caché L1? Responde en el [cuestionario](#): pregunta 12.


Cuando la caché tiene espacio suficiente para albergar el código y datos de todos los programas que se ejecutan, la localidad de los programas tiene poca influencia sobre el tiempo de ejecución de los mismos. Para comprobarlo se van a llevar a cabo nuevas modificaciones sobre el programa `3-3loc1.c`.



- Modifica los programas `3-3loc1.c` y `3-3loc3.c`, cambiando el número de filas y columnas de la matriz para que la matriz ocupe aproximadamente la mitad de la capacidad de la caché L1 de datos. Incrementa además el número de repeticiones para conseguir que el tiempo de ejecución sea de varios segundos. Por ejemplo, puedes incrementar el número de repeticiones en la misma proporción en la que has reducido la capacidad de la matriz. Guarda los programas resultantes con los nombres `3-3loc5.c` y `3-3loc6.c` respectivamente.
- Compila, enlaza, ejecuta dichos programas y compara los tiempos de ejecución. La diferencia, ¿es mayor o menor que en el caso de los programas `3-3loc1.c` y `3-3loc3.c`? Responde en el [cuestionario](#): pregunta 13.

3. Simulación de los accesos a memoria con Valgrind

En este apartado vamos a confirmar que el rendimiento observado en el punto anterior dependiendo de cómo se recorria la matriz tenía que ver con el aprovechamiento de la memoria caché. Para ello, vamos a utilizar el paquete Valgrind, más concretamente su herramienta `cachegrind`, que nos permite simular la traza de accesos a memoria del programa. Para ello, simula la ejecución del programa sobre la misma jerarquía de memoria que implementa el computador.



- Simula la traza de accesos a memoria del programa `3-3loc1` con el siguiente comando. Tardará un tiempo en ejecutarse.

```
$> valgrind --tool=cachegrind ./3-3loc1
```
- Observa los resultados y comprueba cómo la tasa de fallos en la caché es relativamente baja, pues el programa aprovecha la localidad espacial al acceder a la matriz.
- Alguna de las abreviaturas que utiliza `cachegrind` para mostrar los resultados son:
 - `I`: instrucción.
 - `D`: datos (`rd`: lectura, `wr`: escritura).
 - `II`: caché L1 de instrucciones.
 - `DI`: caché L1 de datos.
 - `LL`: resto de cachés de la jerarquía (L2, L3).
 - `LLI`: accesos a instrucción en el resto de cachés de la jerarquía.
 - `LLd`: accesos a datos en el resto de cachés de la jerarquía.
- Prueba ahora a simular la traza de accesos a memoria del programa `3-3loc3`, donde se accedía a la matriz por columnas, desaprovechando completamente la localidad en el acceso a los datos.

```
$> valgrind --tool=cachegrind ./3-3loc3
```
- Comprobarás que la tasa de fallos de caché es prácticamente del 100%, pues todos los accesos a la matriz generan fallos de caché.

Archivos de la práctica

En este apartado vamos a confirmar que el rendimiento observado en el punto anterior dependiendo de cómo se recorria la matriz tenía que ver con el aprovechamiento de la memoria caché. Para ello, vamos a utilizar el paquete Valgrind, más concretamente su herramienta `cachegrind`, que nos permite simular la traza de accesos a memoria del programa. Para ello, simula la ejecución del programa sobre la misma jerarquía de memoria que implementa el computador.

En este apartado vamos a confirmar que el rendimiento observado en el punto anterior dependiendo de cómo se recorria la matriz tenía que ver con el aprovechamiento de la memoria caché. Para ello, vamos a utilizar el paquete Valgrind, más concretamente su herramienta `cachegrind`, que nos permite simular la traza de accesos a memoria del programa. Para ello, simula la ejecución del programa sobre la misma jerarquía de memoria que implementa el computador.

Ejercicios

- Busca información sobre el modelo de CPU de tu ordenador y la de tu teléfono móvil. Determina el número de niveles de caché y su configuración y tamaños.