Práctica 4 - Parte 1: Toma de contacto con Cachegrind

Valgrind es una herramienta disponible en Linux que permite instrumentar un ejecutable con varios propósitos, como detectar errores de memoria u obtener estadísticas sobre el comportamiento de una aplicación. En nuestra asignatura va a ser esta segunda funcionalidad la que nos interese, en concreto mediante la herramienta Cachegrind, que detecta los accesos a memoria de la aplicación y los simula sobre una jerarquía caché cuya configuración podemos controlar. En concreto, simula un sistema con una caché de primer nivel de instrucciones (I1), y otra separada de datos (D1), estando ambas conectadas a una caché unificada de nivel 2 que valgrind denomina Last Level Cache (LL). El manual de Cachegrind se encuentra en `http://valgrind.org/docs/manual/cg-manual.html`. En la plataforma `moodle` también tenéis un breve manual de uso que se entrega en forma de apéndice a esta práctica. `Este apéndice debe leerse con detenimiento antes de realizar esta práctica`

Primero debemos familiarizarnos con el uso de Cachegrind y comprender la influencia que tienen los parámetros de la caché sobre la tasa de fallos, pues ambos conocimientos deben adquirirse antes de intentar mejorar la localidad de los códigos.

1. Comienza por escribir el programa `traspuesta.c`:

```
/* programa traspuesta.c */
#define N 200
double a[N][N], b[N][N];

int main()
{ int i, j;

  for(i = 0; i < N; i++)
    for(j = 0; j < N; j++) {
      a[i][j] = (double)(i * j);
      b[j][i] = a[i][j];
    }

}
```

2. A continuación compílalo con gcc sin utilizar ninguna opción de optimización y analiza su comportamiento caché mediante Cachegrind. Para ello ejecutaremos los siguientes comandos (ver más detalles en Apéndice)

```
gcc -o traspuesta traspuesta.c
valgrind --tool=cachegrind --LL=2048,1,64 ./traspuesta
```

   Anota:

   - El número de fallos obtenidos en la caché de último nivel
   - La tasa de fallos local de la caché de último nivel
   - El número de accesos total a datos
   - La tasa de fallos global del sistema de caché

3. Ahora compílalo con gcc con la opción de optimización `-O2` y analiza su comportamiento caché mediante Cachegrind. Para ello ejecutamos los siguientes comandos (ver más detalles en Apéndice):

```
gcc -O2 -o traspuesta traspuesta.c
valgrind --tool=cachegrind --LL=2048,1,64 ./traspuesta
```

   Anota:

   - El número de accesos total a datos

   Responde:

- ¿Cómo cambia el número de accesos total a datos entre la versión optimizada y la versión sin optimizar? ¿Por qué?

4. Ahora vamos a ver en detalle cuántos fallos caché produce cada instrucción del código. Para ello teclearemos los siguientes comandos (ver más detalles en Apéndice).

```
gcc -g -O2 -o traspuesta traspuesta.c
valgrind --tool=cachegrind --cachegrind-out-file=output --D1=4096,1,64 ./traspuesta
cg_annotate --auto=yes output
```

Anota:

- ¿Cuántos fallos produce la instrucción `a[i][j] = (double)(i * j)` en la caché de datos de nivel 1?
- ¿Cuántos fallos produce la instrucción `b[j][i] = a[i][j]` en la caché de datos de nivel 1?

Responde:

- ¿A qué crees que se debe la diferencia en el número de fallos entre ambas instrucciones?

Práctica 4 - Parte 2: Intercambio de bucles, Fusión de arrays, fusión de bucles.

En esta sesión estudiaremos tres técnicas orientadas a la mejora del rendimiento caché a través de la reestructuración de un código dado.

1. Intercambio de bucles

Algunos programas tienen bucles anidados que acceden a los datos en memoria en un orden no secuencial. Simplemente intercambiando el orden del anidamiento se puede conseguir que el código acceda a los datos en el orden en que están almacenados. Por ejemplo, en FORTRAN el compilador almacena los datos por columnas, con lo que el orden ideal de acceso de los datos es columna a columna. Por el contrario, en C y C++ los datos se almacenan consecutivamente a lo largo de la última dimensión de un array, a continuación a lo largo de la antepenúltima, y así consecutivamente. La Figura 1 ilustra este tipo de mapeado para una pequeña matriz de ejemplo. Puede verse que en el caso de un array bidimensional, los datos de cada fila están almacenados en posiciones consecutivas en memoria, por lo que un acceso por columnas sería contraproducente, puesto que entre cada dos elementos consecutivos en una columna tenemos almacenados todos los de una fila en la memoria. Por tanto, si se encuentra un anidamiento de bucles como el de la Figura 2 en donde una matriz multidimensional no es accedida de forma consecutiva, intercambiar el orden de los bucles para convertir dicho acceso en secuencial puede proporcionar importantes mejoras de rendimiento. En cualquier caso, debe vigilarse que dicho reordenamiento sea legal, es decir, que tras intercambiar los bucles el resultado de la computación siga siendo el mismo.

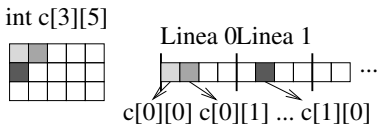

Figura 1: Mapeado de los elementos de una matriz en C a posiciones de memoria

```
for( j=0; j<N; j++)
   for( i=0; i<M; i++)
      x[i][j] = 2 * x[i][j];
```

⇒

```
for( i=0; i<M; i++)
   for( j=0; j<N; j++)
      x[i][j] = 2 * x[i][j];
```

Figura 2: Ejemplo de intercambio de bucles

2. Fusión de arrays

Algunos programas referencian al mismo tiempo dos o más arrays de la misma dimensión usando los mismos índices. En el caso de que las posiciones correspondientes de dichos arrays estuvieran mapeadas a los mismos conjuntos de la caché esto daría lugar una serie de fallos de conflicto sistemáticos. Una situación típica en que da este efecto es en el caso de arrays con un tamaño que es una potencia de dos y se han declarado de forma consecutiva en el programa. La fusión de arrays mezcla múltiples arrays en un único array compuesto. En lenguaje C esto se realiza declarando un array de estructuras en lugar de dos o más arrays, tal como muestra la Figura 3.

```
int val[TAMAÑO];
int key[TAMAÑO];
```

⇒

```
struct miregistro {
  int val;
  int key;
}
struct miregistro union_array[TAMAÑO];
```

Figura 3: Fusionado de arrays

Otra situación, aún más habitual, en la que fusionar arrays es de utilidad, es cuando tenemos dos o más estructuras de datos que se acceden de forma conjunta no secuencial. El acceso no secuencial puede ser

un acceso a través de una indirección mediante un array de enteros (tipo `a[b[i]]`), por punteros, o un acceso regular con una distancia entre cada dos accesos consecutivos superior a la longitud de una línea caché.

3. Fusión de bucles (Técnica a estudiar de forma autónoma por el alumno)

Los programas numéricos a menudo consisten en varias operaciones sobre los mismos datos, codificadas como múltiples bucles sobre los mismos arrays. Los arrays son con frecuencia mayores que las cachés, con lo que normalmente es muy díficil que los datos traidos a las cachés durante la ejecución de un bucle permanezcan en ella cuando se los va a volver a acceder en un bucle subsecuente. En estas ocasiones, cuando el orden de ejecución del programa puede modificarse de forma legal fusionando bucles consecutivos, conseguimos que el reuso de dichos datos en la caché pueda efectuarse con éxito.

```
for( i=0; i<N; i++)
   for( j=0; j<N; j++)
      a[i][j] = 1 / b[i][j] * c[i][j];

for( i=0; i<N; i++)
   for( j=0; j<N; j++)
      d[i][j] = a[i][j] + c[i][j];
```

⇒

```
for( i=0; i<N; i++)
   for( j=0; j<N; j++) {
      a[i][j] = 1 / b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];
   }
```

Figura 4: Ejemplo de fusión de bucles

a) En el moodle tenéis un código escrito en C en el que la aplicación de cada técnica es beneficioso. Aplica la técnica correspondiente sobre cada uno de ellos. El código proporcionado incluye comandos que permiten medir su tiempo de ejecución.

b) Medir el tiempo de ejecución de los códigos en el ordenador que se esté usando del laboratorio. Deberán compararse los tiempos de ejecución del código antes y después de aplicar la optimización. Sugerimos que en las compilaciones se use al menos el flag `-O` o `-O2` para que el compilador aplique algunas optimizaciones básicas.

Recuerda que como la ejecución con Cachegrind es bastante más lenta que la ejecución normal deberás ajustar en tiempo de compilación los tamaños de los bucles y estructuras de datos a través de la macro `N`, y el número de repeticiones de los bucles que se usan para medir los tiempos de ejecución a través de la macro `NUM_REPS`. Ejemplo: `gcc -g -O2 -DN=100 -DNUM_REPS=1 miejemplo.c`.

c) Ejecutar estos códigos con Cachegrind para comparar la tasa de fallos obtenida antes y después de aplicar la optimización. Encontrar una configuración de la caché (tamaño de la caché, tamaño de bloque/línea y asociatividad) para la que se aprecie la mejora obtenida al aplicar cada técnica.

Responder a las preguntas que aparecen en el cuestionario `moodle` asociado a esta sesión de prácticas.