Grupo 08:

- GONZALO JUÁREZ TELLO.
- HODEI URIGOITIA MERODIO.

Laboratorio de caché.

Tarea 1. Fusión de bucles

Configuración general:

- Líneas de caché: 64 bytes.
- Asociatividad: 8 (para todos los niveles de caché).
- Caché de último nivel (LL): 128KiB.

Notas:

El output de cg_annotate al que se hace referencia es el correspondiente a la función de interés en el ejecutable.

Análisis para caché L1D de 64KiB:

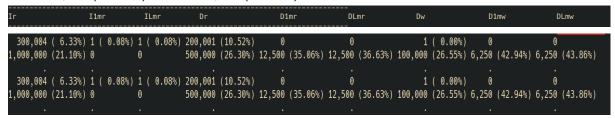
loop_merge

instrucciones:

Ir 2,600,015 (54.85%) - I1mr 2 (0.16%) - ILmr 2 (0.16%)

datos:

Dr 1,400,004 (73.64%) - D1mr 25,000 (70.12%) - DLmr 25,000 (73.26%) - Dw 200,004 (53.11%) - D1mw 12,500 (85.88%) - DLmw 12,500 (87.72%)



1ra línea: inicialización, condición, y acción post-iteración del primer loop.

2da línea: primera suma.

3ra línea: guiones (corchetes en el código fuente).

4ra línea: inicialización, condición, y acción post-iteración del primer loop.

5ta línea: segunda suma.

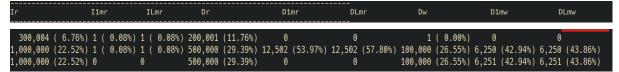
loop_merge_opt

instrucciones:

Ir 2,300,011 (51.80%) - I1mr 2 (0.16%) - ILmr 2 (0.16%)

datos:

Dr 1,200,003 (70.54%) - D1mr 12,503 (53.98%) - DLmr 12,503 (57.80%) - Dw 200,003 (53.11%) - D1mw 12,501 (85.88%) - DLmw 12,501 (87.72%)



1ra línea: inicialización, condición, y acción post-iteración del único loop.

2da línea: primera suma. 3ra línea: segunda suma.

> conclusiones:

 a) loop_merge tiene el doble de misses a caché de datos que loop_merge_opt (luego D1mr == DLmr && D1mw == DLMw para ambos, o sea, estos son fallos en LL y se buscan los datos en memoria principal). b) loop_merge tiene 200 mil accesos más a memoria y 300 mil accesos más a instrucciones que loop_merge_opt, esto es por la línea repetida del loop que se compila a las siguientes 3 líneas de assembler x86 64 (output de gdb).

```
0x000000000040114f <+73>: addl $0x1,-0x4(%rbp)
0x0000000000401153 <+77>: cmpl $0x1869f,-0x4(%rbp)
0x000000000040115a <+84>: jle 0x401121 <main()+27>
```

Como se puede observar, 2 accesos a memoria (add y cmp), 3 instrucciones, son 100 mil iteraciones por solo queda multiplicar para obtener la diferencia ya mencionada.

Análisis para caché L1D de 32KiB y 16KiB:

En lo que concierne a la función de interés, el output casi no nota cambios numéricos. El output puede verse en la carpeta del trabajo práctico en la siguiente carpeta del repositorio de github: https://github.com/juarez-gonza/arcos-uc3m/tree/master/lab-1/lab-cache/vgrind

Tarea 2. Estructuras y arrays

Configuración general:

- Líneas de caché: 64 bytes.
- Asociatividad: 8 (para todos los niveles de caché).
- Caché de último nivel (LL): 256KiB.

Análisis para caché L1D de 32KiB:

soa

instrucciones:

Ir 2,704,716 - I1mr 4 - ILmr 4

datos:

Dr 1,201,177 - D1mr 51,176 - DLmr 51,175 - Dw 200,006 - D1mw 25,003 - DLmw 25,003

```
Auto-annotated source: /home/hodei/Desktop/Universidad/Arquitectura de computadores/Lab C++/Lab 1/lab-cache-2021/soa.cpp
          I1mr ILmr Dr
                              D1mr DLmr
                                             Dw
                                                      D1mw DLmw
                                                                     . constexpr int maxsize = 100000:
                                                                       struct points {
  double x[maxsize];
                                                                          double y[maxsize];
                                                                        int main() {
  points a{}, b{}, c{}; // Default init
                       1,172 1,171 1,170
 400,003
                   1 200,001
                                                                          for (int i=0; i<maxsize; ++i) {</pre>
                                                                            a.x[i] = b.x[i] + c.x[i];
a.y[i] = b.y[i] + c.y[i];
                   0 500,000 25,002 25,002 100,000 12,500 12,500
.000.000
                   1 500,000 25,002 25,002 100,000 12,501 12,501
1.300.000
```

aos

instrucciones:

Ir 4,204,702 - I1mr 3 - ILmr 3

datos:

Dr 1,201,177 - D1mr 51,174 - DLmr 51,173 - Dw 200,004 - D1mw 25,000 - DLmw 24,980

```
Auto-annotated source: /home/hodei/Desktop/Universidad/Arquitectura de computadores/Lab C++/Lab 1/lab-cache-2021/aos.cpp
        Iimr Ilmr Dr
                           D1mr
                                  DLmc
                                        Dw
                                                  D1mw
                                                        DLmw
                                                                  struct point {
                                                                    double x;
                                                                    double y;
                                                                  };
                                                                  int main() {
  constexpr int maxsize = 100000;
                            1,171
                                                                    point a[maxsize], b[maxsize], c[maxsize];
                                                                    for (int i=0; i<maxsize; ++i) {
900,000
                           50,002
                                  50,002
                                         100,000
                                                  24,998 24,978
900,000
                                        0 100,000
```

> conclusiones:

 a) aos tiene casi el doble de accesos a instrucciones que soa, debido a la forma de organizar los datos. El código assembler generado para los loops difiere en el acceso a los elementos.
 Como se puede ver en el output de objdump:

en soa un elemento de array se accede en 3 instrucciones (amd64):

```
      401185:
      8b 45 fc
      mov -0x4(%rbp),%eax

      401188:
      48 98
      cltq

      40118a:
      f2 0f 10 8c c5 f0 2b
      movsd -0x30d410(%rbp,%rax,8),%xmm1

      401191:
      cf ff
```

en rbp-0x4 está la variable "i", luego se extiende el signo de "i", finalmente se indexa el struct de arrays (en rbp-0x30d410).

mientras que en aos un elemento de array se accede en 6 instrucciones (amd64):

```
8b 45 fc
401124:
                                                 -0x4(%rbp),%eax
                                         mov
401127:
               48 98
                                         cltq
401129:
               48 c1 e0 04
                                         shl
                                                 $0x4,%rax
40112d:
               48 01 e8
                                                 %rbp,%rax
                                         add
401130:
               48 2d 10 d4 30 00
                                         sub
                                                 $0x30d410,%rax
401136:
               f2 0f 10
                        08
                                         movsd
                                                 (%rax),%xmm1
```

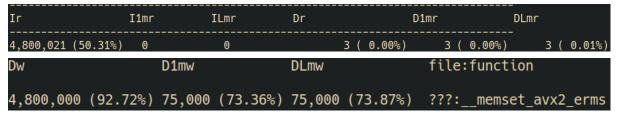
de nuevo rbp-0x4 contiene la variable "i", se extiende el signo de "i", se multiplica "i" por 16 (shift 4) para calcular la dirección de un struct en el array de structs (en rbp-0x30d410) y finalmente se mueve el valor que hay en esa dirección a un registro de punto flotante.

b) Podemos observar también que en aos todos los fallos se llevan a cabo en la primera suma del bucle, a diferencia de en soa, que se llevan a cabo la mitad en la primera suma y la otra mitad en la segunda. Esto es debido a que en aos todo un struct es traído a caché en su primera referencia (primera línea) y para la segunda referencia no produce un fallo (segunda línea).

De hecho, al parecer, en aos se llena una línea de caché de 64 bytes con 4 structs (2 double->16 bytes c/struct) en cada referencia. 75 mil fallos a caché (50 mil lectura + 25 mil escritura), cada fallo llena 64 bytes de caché, 64*(75 mil) = 4.8 millones de bytes, es decir, el tamaño del array de struct multiplicado por 3 (3 * 16 * 100 mil = 1.6 millones de bytes) porque hay 3 arrays.

Mientras que en soa se llenan las líneas de caché de a 8 doubles (8 bytes * 8 = 64 bytes). Primero fallan los accesos a "x". En bytes, (25 mil + 12.5 mil) fallos * 64 bytes = 2.4 millones bytes = 3 * 8 bytes * 100 mil = 3 * tamaño de un array de 100 mil doubles. Luego fallan los accesos a "y" (mitad restante en segunda línea, 37500 fallos más).

c) El compilador puede añadir llamados a memset para 0-inicializar arrays. Ese fue el caso para el binario de soa. El overhead de memset es importante, como se puede ver en las capturas:



Puede ser distinto al usar otras versiones de compilador y/o otras máquinas.

Análisis para caché L1D de 16KiB y 8KiB:

Al igual que en la primera tarea, el output casi no nota cambios numéricos.

El output puede verse en la carpeta del trabajo práctico en la siguiente carpeta del repositorio de github:

https://github.com/juarez-gonza/arcos uc3m/tree/master/lab 1/lab-cache/vgrind